AD_____

LEVEL II

13

FINAL TECHNICAL REPORT
GIT–ICS–81/09
For Period Covering 1 July 1980 – 30 June 1981

PERFORMANCE OF DISTRIBUTED AND DECENTRALIZED CONTROL MODELS FOR FULLY DISTRIBUTED PROCESSING SYSTEMS

Initial Simulation Studies

By

Philip H. Enslow, Jr.
Timothy G. Saponas

Prepared for

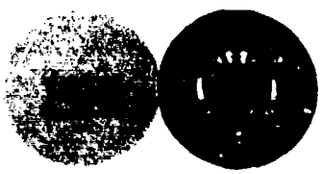
ROME AIR DEVELOPMENT CENTER (ISCP)
DEPARTMENT OF THE AIR FORCE
GRIFFISS AIR FORCE BASE, NEW YORK 13441

Under

Contract Number F30602–78–C–0120
GIT Project Number G36–654

July, 1981

GEORGIA INSTITUTE OF TECHNOLOGY
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332



THE RESEARCH PROGRAM IN
FULLY DISTRIBUTED PROCESSING SYSTEMS

PERFORMANCE OF DISTRIBUTED AND DECENTRALIZED CONTROL MODELS

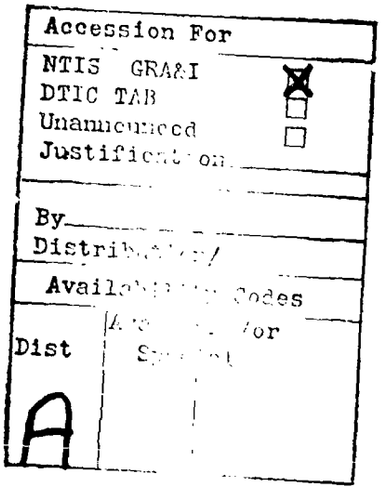
FOR

FULLY DISTRIBUTED PROCESSING SYSTEMS

Initial Simulation Studies

FINAL TECHNICAL REPORT

GIT-ICS-81/09



1 July 1980 - 30 June 1981

Philip H. Enslow, Jr.
Timothy G. Saponas

July, 1981

Rome Air Development Center (ISCP)
Department of the Air Force
Griffiss Air Force Base, New York 13441

Contract Number F30602-78-C-0120
GIT Project Number G36-654

The Georgia Tech Research Program in
Fully Distributed Processing Systems
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE AIR FORCE POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER GIT-ICS-81/09	2. GOVT ACCESSION NO. AD-A108 974	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Performance of Distributed and Decentralized Control Models for Fully Distributed Processing Systems --- Initial Simulation Studies		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 1 July 80 - 30 June 81
		6. PERFORMING ORG. REPORT NUMBER GIT-ICS-81/09
7. AUTHOR(s) Philip H. Enslow Jr. Timothy G. Saponas		8. CONTRACT OR GRANT NUMBER(s) F30602-78-C-0120
9. PERFORMING ORGANIZATION NAME AND ADDRESS School of Information and Computer Science Georgia Institute of Technology Atlanta, Georgia 30332		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCP) Department of the Air Force Griffiss Air Force Base, New York 13441		12. REPORT DATE July, 1981 13. NUMBER OF PAGES 96 + x
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) same as item 11		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE n/a

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution limited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Thomas F. Lawrence (ISCP)
The view, opinions, and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Air Force position, policy, or decision, unless so designated by other documentation.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Control
Decentralized Control
Distributed Processing
Fully Distributed Processing Systems
Network
Network Operating System

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

An essential component of a Fully Distributed Processing System (FDPS) is the distributed and decentralized control. This component unifies the management of the resources of the FDPS and provides system transparency to the user. In a previous study, the problems of distributed and decentralized control were analyzed resulting in the specification of several control models. This study continues that work by further specifying the control models defined in the first report and comparing the performance of these models in various environments. This performance analysis is accomplished by means of

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

4/1.1.74

simulation experiments. The results of the experiments indicate that the control message traffic generated by the distributed and decentralized control is much less than expected and probably does not present a barrier to the implementation of FDPSs. Comparison of the results of the simulation of a uniprocessor and that of an FDPS indicate that little or no loss of performance is experienced by the FDPS. An important limitation of these initial performance studies is the fact that user traffic is not included in this series of tests.

PREFACE

The simulation results that are discussed in this report represent the findings obtained during the period of the grant, 1 July 1980 - 30 June 1981. Further simulation studies have been conducted and will be documented in a Ph.D. thesis by Timothy G. Saponas entitled "Distributed and Decentralized Control in a Fully Distributed Processing System," which is to be published in the near future.

ABSTRACT

An essential component of a Fully Distributed Processing System (FDPS) is the distributed and decentralized control. This component unifies the management of the resources of the FDPS and provides system transparency to the user. In a previous study, the problems of distributed and decentralized control were analyzed resulting in the specification of several control models. This study continues that work by further specifying the control models defined in the first report and comparing the performance of these models in various environments. This performance analysis is accomplished by means of simulation experiments. The results of the experiments indicate that the control message traffic generated by the distributed and decentralized control is much less than expected and probably does not present a barrier to the implementation of FDPSs. Comparison of the results of the simulation of a uniprocessor and that of an FDPS indicate that little or no loss of performance is experienced by the FDPS. An important limitation of these initial performance studies is the fact that user traffic is not included in this series of tests.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

SECTION 1

INTRODUCTION

Distributed Processing Systems are currently receiving a very large amount of attention. This is due in part to the claims that these systems will provide a number of advantages over contemporary systems (see Table 1). Some of the more important potential advantages being publicized are the following: increased performance (with respect to both throughput and response time), ability to share resources, ease of system expansion, and the ability to provide fault-tolerance.

Table 1. "Benefits" Provided by Distributed Processing Systems

A Representative List Assembled from Claims Made in Actual Sales Literature

High Availability and Reliability
Reduced Network Costs
High System Performance
Fast Response Time
High Throughput
Graceful Degradation, Fail-soft
Ease of Modular and Incremental Growth
Configuration Flexibility
Automatic Load and Resource Sharing
Easily Adaptable to Changes in Workload
Incremental Replacement and/or Upgrade
Easy Expansion in Capacity and/or Function
Good Response to Temporary Overloads

This report is concerned with a particular class of distributed proces-

sing systems, "Fully Distributed Processing Systems (FDPS)," which are the focus of a major research program at the Georgia Institute of Technology. For a system to be classified as an "FDPS," it must possess all five of the following characteristics:

1. Multiplicity of resources: an FDPS is composed of a multiplicity of "general-purpose" resources that can be freely assigned on a short-term basis to various system tasks as required (e.g., hardware and software processors, shared data bases, etc.).
2. Component interconnection: the active components in the FDPS are physically connected by a communication network(s) utilizing two-party, cooperative protocols to control the physical transfer of data (i.e., loose physical and logical coupling).
3. Unity of control: the executive control of an FDPS must define and support a unified set of policies governing the operation and utilization of all physical and logical resources.
4. System transparency: users must be able to request services by generic names without being aware of their physical location or even the fact that multiple copies of the resources may exist. (System transparency is designed to aid rather than inhibit and, therefore, can be overridden. A user who is concerned about the performance of a particular application can provide system-specific information to aid in the formulation of management control decisions.)
5. Component autonomy: both the logical and physical components of an FDPS should interact in a manner described as "cooperative autonomy" [Ens178]. This means that the components operate in an autonomous fashion requiring cooperation among processes for the exchange of information as well as for the provision of services. In a cooperatively autonomous control environment, the components are afforded the ability to refuse requests for service, regardless of whether the service request involves execution of a process or the use of a file. This could result in anarchy except for the fact that all components adhere to a common set of system utilization and management policies expressed by the philosophy of the executive control.

A more detailed explanation of these characteristics is found in Section 2 of this report.

An essential component of an FDPS is the distributed and decentralized control. This component unifies the management of the resources of the FDPS and provides system transparency to the user. A previous study (see [Ens181]) examined the characteristics of various models of distributed and decentralized control that met this criteria and identified a number of

variations possible in specific features of the different models. That research helped to define more clearly the exact nature of the operation of an FDPS, the problems inherent in distributed and decentralized control, and possible solutions to these problems.

The scope and goal of the present work is to both qualitatively and quantitatively evaluate the effect of these features on the performance of the various models of control. The qualitative evaluation is intended to demonstrate how a particular model performs in a specific environment. In this phase, the validity of a model is established. The quantitative evaluation, on the other hand, is intended to examine in general the relative merits of decentralized control and provide data to support conclusions about the relative performance of the various models.

SECTION 2

BACKGROUND

2.1 THE DEFINITION OF AN FDPS

Fully Distributed Processing Systems (FDPS) were first defined by Enslow in 1976 [Ensl78] although the designation "fully" was not added until 1978 when it became necessary to clearly distinguish this specific class of systems from the many others being presented as "distributed processing systems." As discussed in Section 1, an FDPS is distinguished by the following characteristics:

1. Multiplicity of resources.
2. Component interconnection.
3. Unity of control.
4. System transparency.
5. Component autonomy.

It is important to note that in order for a system to qualify as being fully distributed it must possess all five of the criteria presented in this definition.

2.1.1 Multiple Resources and Their Utilization

The requirement for resource multiplicity concerns the assignable resources that a system provides. Therefore, the type of resources requiring replication depends on the purpose of a system. For example, a distributed system designed to perform real-time computing for air traffic control requires a multiplicity of special-purpose air traffic control processors and display terminals. It is not required that replicated resources be exactly homogeneous; instead, they must be capable of providing the same services.

In addition to the requirement for multiplicity, the system resources must be dynamically reconfigurable to respond to component failures as well as changes in the work load presented to the system. This reconfiguration must occur within a "short" period of time so as to maintain the functional capabilities of the overall system without affecting the operation of components not directly involved. Under normal operation, the system must be able to dynamically assign its tasks to components distributed throughout the system.

The extent to which resources are replicated can range from those systems where none are replicated (not a fully distributed system) to systems with all assignable resources replicated. In addition, the number of copies of a particular resource can vary depending on the system and type of resource. In general, the greater the degree of replication, particularly of resources in high demand, the greater the potential for attaining benefits such as increased performance (response time and throughput), availability, reliability, and flexibility [Ensl78].

2.1.2 Component Interconnection and Communication

The extent of physical distribution of resources in distributed systems can range from the length of a connection between components on a single integrated chip to the distance between two computers communicating through an international network. In addition, interconnection subsystem organizations can vary from a single time-shared bus to a complex, mesh interconnection network. Since a component in a distributed system communicates with other components through its own logical process, all physical and logical resources can be thought of as processes, and interactions between resources can be referred to as interprocess communication [Davi79]. For example, application program interaction with data files is accomplished through communication between logical processes, the application process and the file process.

In an FDPS, both the physical and logical coupling of the system components are characterized as "extremely loose." "Gated" or "master-slave" control of physical transfers is not allowed. Communication (i.e., the physical transfer of messages) is accomplished through the active cooperation and participation of both the sender and addressees. The primary requirement of the interconnection subsystem is that it support such a two-party cooperative protocol. This is essential to enable the system's resources to exist with "cooperative autonomy" at the physical level.

The advantages of using a message-based (loosely-coupled) communication system with a two-party cooperative protocol include reliability, availability, and extensibility. The disadvantage is the additional overhead of message processing incurred to support this method of communication. There are a variety of interconnection organizations and communication techniques that can be used to support a message-based system with a two-party

cooperative protocol.

2.1.3 Unity of Control

In a fully distributed data processing system, individual processors will control local resources with their own local operating systems, which may or may not be unique. As a result, control is distributed throughout the system to control system components that operate autonomously. However, to gain the benefits of distributed processing, it is required that the autonomous components of the system cooperate with each other to achieve the overall objectives of the system. To insure this, the concept of a high-level operating system was created to integrate and unify, at least conceptually, the decentralized control of the system.

A high-level operating system is essential to the successful implementation of a distributed processing system. The high-level operating system is not a centralized block of code exercising strong hierarchical control over the system; instead, it is a well-defined set of policies governing the integrated operation of the system as a whole. To insure reliable and flexible operation of the system, these policies should be implemented with minimal binding to any of the system's components [Ensl78].

What policies are required and how they should be implemented depends greatly on the system. For example, if it is a general-purpose system supporting interactive users, then a command interpreter and a user control language is required to make the system's components compatible and transparent to the user.

2.1.4 Transparency of System Control

The high-level operating system also provides the user with an interface to the distributed system. As a result, the user is accessing the system as a whole rather than just a single computer in the network.

In order to increase the effectiveness of the distributed system, the actual system organization is made transparent. The user is presented with a virtual machine and a command language to access it. Using this command language, the user requests services by name and does not need to specify the specific server to be used. Clearly, multiple requests for the same service might be assigned to different servers depending on the state of the total system when the request is made. However, to make the system truly effective

for all users, knowledgeable individuals must be able to interact with the system more directly, requesting specific servers or developing service routines to increase the efficiency or effectiveness of the system [Ensl78].

2.1.5 Cooperative Autonomy

Cooperative autonomy has already been described at the physical interconnection level. It is also required that all resources be autonomous at the logical control level. A resource must have complete control in determining which requests it will service and what future operations it will perform. However, a resource must also cooperate with other resources by operating according to the policies of the high-level operating system. Cooperative autonomy is an essential prerequisite for systems to have fault tolerance and high degrees of extensibility [Ensl78]. It is perhaps the most important and most distinguishing characteristic of a fully distributed processing system.

2.2 CHARACTERIZATION OF DISTRIBUTED AND DECENTRALIZED CONTROL

2.2.1 General Nature of FDPS Executive Control

The executive control is responsible for managing the resources of the FDPS. Its charter is to perform the management function in such a manner that the resources of the FDPS are unified and users of the FDPS are shielded from the physical realities of distribution. In other words, the executive control provides system transparency for the user.

The executive control of an FDPS can be implemented in many different ways. It can consist of identical modules replicated on all nodes of the system. Alternatively, it can consist of several unique modules distributed in some manner about the system. The essential point is that the term "executive control" does not necessarily mean a particular module at a particular node, but rather the entire collection of modules that are distributed somehow throughout the system and are working together to manage the system's resources.

2.2.2 Control Problems Resulting from the FDPS Environment

Several characteristics of an FDPS are found to directly impact the design and implementation of the executive control. These include system transparency to the user, extremely loose physical and logical coupling, and cooperative autonomy as the basic mode of component interaction. System

transparency means that the FDPS appears to a user as a large uniprocessor which has available a variety of services. It must be possible for the user to obtain these services by naming them without specifying any information concerning the details of their physical location. The task of locating all appropriate instances (copies) of a particular resource and choosing the instance to be utilized is left to the executive control.

"Cooperative autonomy" is another characteristic of an FDPS that has a large effect on the design of the executive control. The "lower-level" control functions of both the logical and physical resource components of an FDPS are designed to operate in a "cooperatively autonomous" fashion. Thus, the executive control must be designed such that any resource is able to refuse a request even though it may have physically accepted the message containing that request. Degeneration into total anarchy is prevented by the establishment of a common set of criteria to be followed by all resources in determining whether a request is accepted and serviced as originally presented, accepted only after bidding or negotiation, or rejected.

Another important FDPS characteristic that definitely affects the design of its executive control is the extremely loose coupling of both physical and logical resources. The components of an FDPS are connected by communication paths of relatively low bandwidth. The direct sharing of primary memory between processors is not acceptable. Even though the logical coupling could still be loose with this physical interconnection mechanism, the presence of a single critical hardware element, the shared memory, would create fault-tolerance limitations. Therefore, all communication takes place over "standard" input/output paths. The actual data rates that can be supported are primarily a function of the interconnections between the processors and the capability of their input/output paths. The available transfer rates are much less than memory transfer rates. This implies that the sharing of control information among components on different processors is greatly restricted. System control is forced to work with information that is "out-of-date" and, as a result, perhaps "inaccurate."

The control of an FDPS requires the participation and cooperation of components at all layers of the system. This implies that there are elements of FDPS control present in the lowest levels of the hardware and software com-

ponents. This study is primarily interested in the software components of the FDPS control which are typically referred to as "the executive control." Low-level aspects of FDPS control will not be directly examined.

The executive control is responsible for managing the physical and logical resources of a system. It accepts user requests and obtains and schedules the resources necessary to satisfy a user's needs. The manner in which these tasks are accomplished is designed to unify the distributed components of the system into a whole and provide system transparency to the user.

2.2.3 Why Not Centralized Control?

Why is a centralized method of control not appropriate? In systems utilizing a centralized executive control, all of the control processes share a single, coherent, and accurate view of the entire system state. An FDPS, though, contains only loosely-coupled components, the communication between which is limited and subject to variable time delays. This means that one cannot guarantee that all control processes will have the same view of the system state [Jens78]. In fact, it is a significant characteristic of an FDPS that all control processes will probably not have a consistent view.

A centralized executive control weakens the fault-tolerance of the overall system due to the existence of a single critical element, the executive control component itself. This obstacle, though, is not insurmountable. Strategies do exist for providing fault-tolerance in centralized applications. Garcia-Molina [Garc79], for example, has described a scheme for providing fault-tolerance in a distributed data base management system with a centralized control. Approaches of this type typically assume that failures are extremely rare events and that the system can tolerate the dedication of a relatively long interval of time to reconfiguration. These restrictions may be unacceptable in an FDPS environment in which it is important to provide fault-tolerance with a minimum of disruption to the services being supported.

Also, the extremely important issue of overall system performance must be considered. A distributed processing system is expected to utilize a large quantity and a wide variety of resources. If a completely centralized executive control is implemented, there is a high probability that a

bottleneck will be created in the node executing the control functions. A distributed and decentralized approach to control attempts to remove this bottleneck by dispersing the control decisions among multiple components on different nodes.

2.2.4 Distributed vs. Decentralized

The discussion above supports the requirement that the executive control of an FDPS must be both "distributed" and "decentralized," and it should be noted that there is a clear distinction between the terms "distributed control" and "decentralized control" as they are used in the context of this project. "Distributed control" is characterized by having its executing components physically located on different nodes. This means there are multiple loci of control activity. In "decentralized control," on the other hand, control decisions are made independently by separate components. In other words, there are multiple loci of control decision making. Thus, distributed and decentralized control has active components located on different nodes, and those components are capable of making independent control decisions.

2.2.5 Rationale Behind Distributed and Decentralized Control

The reasons for distributing and decentralizing control result from two basic goals of an FDPS, to improve performance and to provide a more fault-tolerant system. With decentralized decision making, a system can potentially provide responses to requests in a shorter amount of time due to the increased utilization of resources which is achieved through the concurrent execution of the decentralized decision makers.

By physically distributing components, one is assured that a system retains the potential to keep running even though some parts have been lost. The ability to function independently of the lost components is provided by decentralized decision making. Thus, by distributing components and decentralizing decision making, the potential for fault-tolerant operation is provided.

2.3 EVALUATION PLAN

The steps performed in the evaluation of the models of control are as follows:

1. Prepare detailed definitions of the models of control.
2. Construct an FDPS simulator.
3. Perform the simulation experiments.
4. Validate the control models.
5. Compare the relative performance data for the different control models.

2.3.1 Definition of Control Models

The first step in the evaluation process is to define in greater detail the models of control originally described in [Ens181]. One of the goals of the present research is to validate the control models in order to examine their performance in certain environments. By looking at the finer details of the models, significant control problems have been discovered which were not apparent from earlier high level studies.

To accomplish this detailed study, the models are translated into a high level programming language, Pascal. The resulting code is presented in Appendix 1 in the form of pseudo code. The pseudo code is derived from the actual Pascal code and is presented in place of the actual code in order to conserve space.

2.3.2 Construction of an FDPS Simulator

In order to perform both validation and performance analysis it is necessary to construct an FDPS simulator. The models of control are translated into Pascal, and the resulting code is incorporated into the simulator. Validation is accomplished by constructing various test cases which are designed to exercise the particular executive control functions being tested. A detailed transaction log is maintained in order to follow the actions of the simulator, and, thus, verify the correct or incorrect performance of each portion of the executive control.

The simulator also collects various performance measurements. These are processed at the termination of the experiment in order to generate performance statistics. The interval during which measurements are collected is user controllable. This allows one to measure steady state values as well as performance during startup.

2.3.3 Simulation Experiments

Simulation experiments are conducted in two phases. The first phase is designed to validate the various models of control. In these experiments, there is no need to collect performance measurements; instead, a detailed log of the simulator's actions is maintained. This is then analyzed in order to observe the behavior of the control model under test.

In the second phase of experiments, performance measurements are collected, but no transaction log is maintained. These experiments are used to obtain data concerning the relative performance of the various models of control. In order to obtain steady state data. measurements are not collected until some time after startup. Several simulations are performed on each model of control. Each simulation provides the control with a different environment. To obtain different environments, the interconnection topology and the bandwidths of the communication links are varied.

The load for the simulator is generated in the following manner. The user specified configuration determines the number of nodes, the connectivity of these nodes, the number of terminals attached to each node, and the initial state of the file system. The file system includes data files, command files, and object files. Each object file specifies a script of actions to be simulated in order to simulate the execution of a user process. The user of the simulator provides a series of commands that can originate from a terminal. These commands form a population of commands from which the load generator randomly selects commands for arrival from specific terminals. The time of command arrival is determined by generating a random number from a particular interval marked by a minimum and a maximum time delay between submission of commands.

2.3.4 Validation of Control Models

Validation of the models of control is achieved by constructing input scripts designed to excercise the particular executive control being tested. The resulting transaction log is analyzed to insure the correct performance of the executive control.

2.3.5 Comparison of the Relative Performance of the Models

After each test, the data reduction portion of the simulator utilizes the performance measurements gathered during the specified interval of time to

compute the following statistics:

1. The average service time for a user session, for a work request, and for a process. (This is computed for all nodes and also averaged over all nodes.)
2. The average response time for a user session, for a work request, and for a process. (This is computed for all nodes and also averaged over all nodes.)
3. The throughput for user sessions, for work requests, and for processes. (This is computed for all nodes and also averaged over all nodes.)
4. For the READY QUEUE on each node, the MESSAGE BLOCKED QUEUE on each node, each DISK WAITING QUEUE on each node, and each LINK QUEUE on each node the following statistics are compiled:
    a. The minimum time spent by a process in the queue.
    b. The maximum time spent by a process in the queue.
    c. The average time spent by a process in the queue.
    d. The minimum queue length observed by a process entering the queue.
    e. The maximum queue length observed by a process entering the queue.
    f. The average queue length observed by a process entering the queue.
5. The number of user messages, control messages, and the total number of messages sent from each node to every other node.
6. The number of user messages, control messages, and the total number of messages sent on each link.

Utilizing these statistics, conclusions concerning the relative merits of each of the models of control are made.

2.4 PROJECT SCOPE AND ORGANIZATION OF THIS REPORT

Following these first two sections of introductory remarks, this paper examines in finer detail the models initially presented in [Ensl81]. Section 3 contains a description of the more important features of the control models under examination. A pseudo code description of these models is provided in Appendix 1.

The simulator used in the evaluation of the models is the topic of discussion in Section 4. In this section, the goals of the simulation experiments, requirements for the simulator, and the structure of the simulator are discussed.

In Section 5, the results of the simulation experiments are examined. This includes discussions of both the validity of the models in certain environments and the relative performance of the various models of control.

Conclusions about the results of the evaluation studies are presented in Section 6. The results of these experiments are summarized and placed into proper perspective and further questions that this study stimulated but failed to answer are identified.

SECTION 3

MODELS OF CONTROL

This research considers six different models of control. These models are described in general terms in this section, and pseudo code for the models is provided in Appendix 1. The models are similar in many respects differing usually only in some particular aspect of control. Therefore, only the first model is presented completely. The others are described by indicating how they differ from the first model.

3.1 THE XFDPS.1 CONTROL MODEL

The XFDPS.1 control model was first defined in [Sapo80] and further refined in [Ensl81]. With the aid of a simulation environment, this model has been even more completely defined. The XFDPS.1 model is composed of six types of components: TASK SET MANAGERs, FILE SYSTEM MANAGERs, FILE SET MANAGERs, PROCESSOR UTILIZATION MANAGERs, PROCESSOR UTILIZATION MONITORs, and PROCESS MANAGERs. (See Figure 1.) The basic strategy of this model of control is to partition the system's resources and assign separate components to manage each partition.

3.1.1 Task Set Manager

A TASK SET MANAGER is assigned to each user terminal as well as to each executing command file. The name TASK SET MANAGER results from the nature of user work requests which originate from user terminals and command files. The work requests specify one or more executable files called tasks (these contain either object code or commands) and any input or output files used by the tasks. It is possible for the tasks of a work request to communicate, and this communication (task connectivity) is also described by the work request. Therefore, each work request specifies a set of tasks, and it is the job of the TASK SET MANAGER to control the execution of that set of tasks.

When a work request arrives, the TASK SET MANAGER parses the work request and initiates construction of the task graph for this work request. In XFDPS.1, only a single copy of the task graph is maintained. This copy is stored at the node where the TASK SET MANAGER for the work request resides. At this stage of work request processing, the task graph contains the initial resource requirements for the work request.

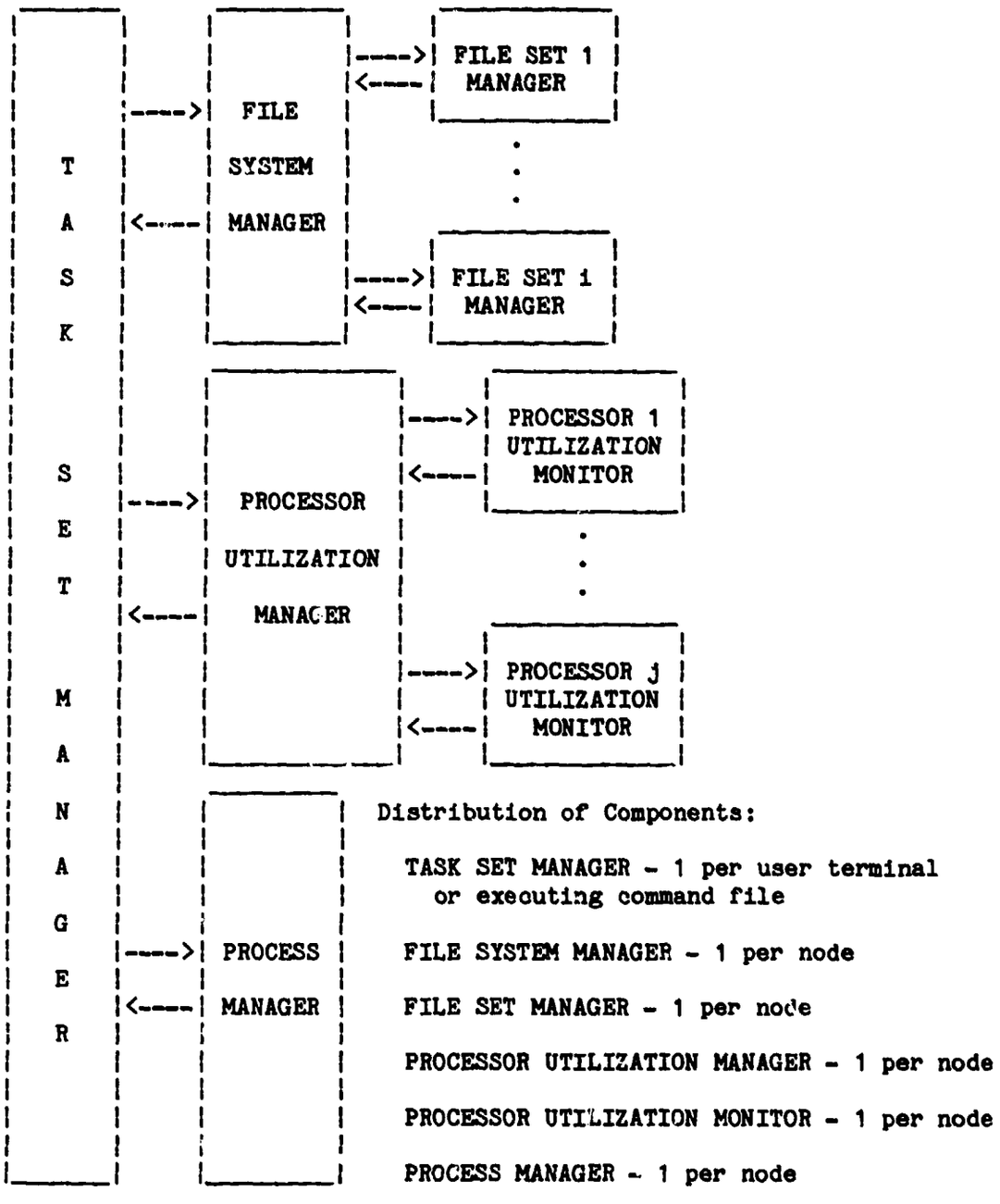


Figure 1. The XFDPS.1 Model of Control

In the next step, a message is sent to the FILE SYSTEM MANAGER residing on the same node as the TASK SET MANAGER requesting file availability information concerning the files needed by the work request. A message is also sent

to the PROCESSOR UTILIZATION MANAGER residing on the same node as the TASK SET MANAGER requesting processor utilization information. This includes the latest utilization information that this particular node has obtained from all other nodes.

When the file availability information and processor utilization information arrive, a work distribution and resource allocation decision is made by the TASK SET MANAGER. At this point, specific files are chosen from the list of files found available and specific processors are chosen as sites for the execution of the various tasks of the work request's task set. In this study no attempt is made to investigate different strategies for distributing work; instead, a single strategy is used for all experiments. (Other work in progress in the FDPS Research Program at Georgia Tech is examining the complete area of work distribution and resource allocation.) In this strategy, a process is assigned to execute on the same node that its object code resides. Data files are not moved but accessed from the node on which they originally resided.

Once the allocation decision is made, a request for the locking of the chosen files is sent by the TASK SET MANAGER to the FILE SYSTEM MANAGER residing on the same node as the TASK SET MANAGER. The desired type of access (READ or WRITE) is also passed along with the lock request. Multiple readers are permitted, but readers are denied access to files already locked for writing, and writers are denied access to files locked for reading or writing. If the FILE SYSTEM MANAGER informs the TASK SET MANAGER that all the desired files have been successfully locked, execution of the work request can be initiated. If the locking operation is not successful, the work request is aborted, and the necessary cleanup operations are performed. The next step after successful file allocation is to send a series of messages to the PROCESS MANAGERs on the various nodes that have been chosen to execute the tasks of the task set informing them that they are to execute a specific subset of tasks.

When a task terminates, its PROCESS MANAGER reports back to the TASK SET MANAGER and indicates the reason for the termination (normal or abnormal). When an indication of an abnormal termination is received, the remaining active tasks of the task set are terminated.

After all tasks of a task set have terminated, one of three possibl actions occurs. If the source of commands is a user terminal, the user is prompted for a new command. If the source is a command file, the next command is obtained. Finally, if the source is a command file and all the commands have been executed, the TASK SET MANAGER is deactivated and the PROCESS MANAGER on the node where the command file was being executed is informed of the termination of the command file.

3.1.2 File System Manager

Replicated on each node of the system is a component called the FILE SYSTEM MANAGER. This module handles the file system requests from all of the TASK SET MANAGERs including requests for file availability information and requests to lock or release files. FILE SYSTEM MANAGERs do not possess any directory information. Therefore, to locate a file, it is necessary that all nodes are queried as to the availability of the file.

The FILE SYSTEM MANAGER satisfies the requests by consulting with the FILE SET MANAGERs (see Section 3.1.3) located on each node of the system. For example, when the FILE SYSTEM MANAGER receives a request for file availability information, messages are prepared and sent to all FILE SET MANAGERs. The FILE SYSTEM MANAGER collects the responses, and when responses from all FILE SET MANAGERs have been obtained, it reports the results to the TASK SET MANAGER which made the request. Requests for the locking or releasing of files are handled in a similar manner.

3.1.3 File Set Manager

The files residing on each node of the system are managed separately from the files on other nodes by a FILE SET MANAGER that is dedicated to managing that set of files. The duties of the FILE SET MANAGER include providing file availability information to inquiring FILE SYSTEM MANAGERs and reserving, locking, and releasing files as requested by FILE SYSTEM MANAGERs. It should be noted that a side effect of gathering file availability information is the placement of a reservation on a file that is found to be available.

3.1.4 Process Utilization Manager

Also present on each node is another component of the executive control, the PROCESSOR UTILIZATION MANAGER. This module is assigned the task of col-

lecting and storing processor utilization information which is obtained from the PROCESSOR UTILIZATION MONITORs (see Section 3.1.5) residing on each of the nodes. When a TASK SET MANAGER asks the PROCESSOR UTILIZATION MANAGER for utilization information, the PROCESSOR UTILIZATION MANAGER responds with the data available at the time of the query.

3.1.5 Processor Utilization Monitor

Each node of the system also has a PROCESSOR UTILIZATION MONITOR that is responsible for collecting various measurements needed to arrive at a value describing the current utilization of the processor on which the PROCESSOR UTILIZATION MONITOR resides. The processor utilization value is periodically transmitted to the PROCESSOR UTILIZATION MANAGERs on all nodes.

3.1.6 Process Manager

Residing on each node of the system is a PROCESS MANAGER whose function is to supervise the execution of processes executing on the node on which it resides. The PROCESS MANAGER is responsible for activating and deactivating processes. If the execution file for a process is an object file, the PROCESS MANAGER will load the object file into memory. This file may reside either locally or on a distant node. If the execution file is a command file, the PROCESS MANAGER sees that a TASK SET MANAGER is activated to respond to the commands of that command file. The PROCESS MANAGER is also responsible for handling process termination. This involves releasing local resources held by the process and informing the TASK SET MANAGER that requested the execution of the process as to the termination of the process.

3.1.7 File Process

In order to provide file access in a manner that is uniform with the operation of the rest of the system, another type of control process is utilized, the FILE PROCESS. For each access to a file, an instance of a FILE PROCESS is created. Therefore, if process "A" is accessing file "X" and process "B" is also accessing file "X", there will be two instances of a FILE PROCESS, each responsible for a particular access to file "X". Communication between FILE PROCESSes and user processes (file reads and writes) or between FILE PROCESSes and PROCESS MANAGERs (loading of object programs) is handled in the same manner as communication between user processes.

3.2 THE XFDPS.2 CONTROL MODEL

The XFDPS.2 model of control differs from the XFDPS.1 model in the manner in which file management is conducted. In this model a centralized directory is maintained. In Appendix 1 the component named FILE SYSTEM MANAGER maintains this directory. This component resides on only one node, the node where the file system directory is maintained. TASK SET MANAGERs communicate directly with this component in order to gain availability information, lock files, or release files.

When a file is locked it is necessary to create a FILE PROCESS in order to provide access to the file. To accomplish this task, the FILE SYSTEM MANAGER sends a message to the node where the file resides requesting activation of a FILE PROCESS providing access to the file. Once this process is created, the FILE SYSTEM MANAGER is given the name of the FILE PROCESS which it then returns to the TASK SET MANAGER that requested the file lock.

3.3 THE XFDPS.3 CONTROL MODEL

In the XFDPS.1 model of control a search for file availability information encompassing all nodes is conducted for each work request. Obtaining this global information is important when one is attempting to obtain optimal resource allocations. In those instances where this is not important a slight variation on the search strategy may be utilized. This strategy is the distinguishing feature of the XFDPS.3 model of control.

Instead of immediately embarking on a global search, a search of local resources (i.e., resources that reside on the same node where the work request originated) is conducted. If all of the required resources are located, no further searches are conducted, and the operations of locking files, activating process, etc., described for model XFDPS.1 are executed. If on the other hand all required resources could not be found, the strategy of model XFDPS.1 is utilized.

3.4 THE XFDPS.4 CONTROL MODEL

The XFDPS.4 model of control utilizes a file management strategy similar to that of the ARAMIS Distributed Computer System [Caba79a,b] in which multiple redundant file system directories are maintained on all nodes of the

system. However, since detailed information about the system described in [Caba79a,b] is not available, model XFDPS.4 cannot be claimed to be an accurate model of that system.

To preserve the consistency of the redundant copies of the file system directory and to provide mutually exclusive access to resources, the following steps are taken. A control message, the control vector (CV), is passed from node to node according to a predetermined ordering of the nodes. The holder of the CV can either release, reserve, or lock files. Therefore, each node collects file system requests and waits for the CV to arrive. Once in possession of the CV, a node can perform the actions necessary to fulfill the requests it has collected.

The modifications to the file system directory are then placed into a message called the update vector (UPV) which is passed to all nodes in order to bring all copies of the file system directory into a consistent state. When the UPV returns to the node holding the CV, all updates have been recorded, and the CV can be sent on to the next node.

3.5 THE XFDPS.5 CONTROL MODEL

In the XFDPS.5 model, files are not reserved when the initial availability request is made, and they are locked only after the work distribution and resource allocation decision has been made. This strategy leads to the possibility of generating an allocation plan that is impossible to carry out if a file chosen for allocation has been given to another process during the interval in which the resource allocation decision is made. In the previous models, the executive control is assured of an allocation being accepted, assuming no component fails.

3.6 THE XFDPS.6 CONTROL MODEL

In the XFDPS.1 model, the task graph for a particular work request is maintained as a single unit and stored on only one node, the node at which the work request originates. The XFDPS.6 model of control utilizes a slightly different strategy. The task graph is constructed on a single node, but once a work distribution and resource allocation decision has been made, portions of the task graph are sent to various nodes. Specifically, those nodes chosen to execute the various tasks of the task graph are given that portion of the

task graph for which they are responsible. Each node, then, must activate the tasks assigned to it and collect termination information concerning those tasks. When all tasks assigned to a particular node have terminated, the node where the work request originally arrived is informed of their termination. One can view this strategy as a two-level hierarchy.

SECTION 4

THE SIMULATOR

In order to obtain quantitative information concerning the relative performance of the various models of control, simulation experiments are conducted. The goals of these experiments are to validate the models of control described in Section 3 and gather data on their relative performance. In order to be able to express the differences between the various models, it is necessary that the simulator provide for the specification of relatively low level features of the control models.

4.1 REQUIREMENTS FOR THE SIMULATOR

The goals described above necessitate the establishment of several requirements for the simulator. In order to handle low level control problems and document solutions to these problems, the control models must be defined in a language capable of clearly expressing the level of detail required at this stage of design. Because a number of models are to be tested, it is important that the coding effort for these models be minimized.

It is expected that the architecture of the network as well as that of individual nodes in the network will affect the relative performance of various control models. Therefore, one must be able to easily modify various architectural attributes. This includes network connectivity, network link capacities, and the capacities and processing speeds of the individual nodes of the network.

Validation of control models is one of the primary goals of the simulation studies. To achieve this goal the simulator must provide the ability to establish specific system states. In other words, specific detailed instances of work requests need to be constructed along with the establishment of specific resource states (e.g., one must be able to set up a series of files in specific locations). These capabilities allow one to exercise specific features of the control models.

The simulation studies also provide performance information. The simulator must utilize a technique for generating work requests reflecting specific distributions. It also needs to collect a variety of performance

measurements and generate appropriate statistical results.

4.2 THE STRUCTURE OF THE SIMULATOR

The simulator is event based and programmed in Pascal. It simulates the hardware components of an FDPS, functions typically provided by local operating systems, functions provided by a distributed and decentralized control, and the load placed upon the system by users attached to the system through terminals.

4.2.1 Architecture Simulated

The hardware organization that is simulated is depicted in Figure 2. The complete system consists of a number of nodes connected by half-duplex communication links. Each node contains a CPU, a communications controller, and perhaps a number of disks. Connected to each node are a number of user terminals. The disk simulation is such that no actual information is stored; only the delays experienced in performing disk input/output are considered. User interprocess communication (IPC) is simulated with time delays but no exchange of real data takes place. However, IPC between components of the executive control involves both simulation of the time delays involved in message transfer and the actual transfer of control information to another simulated node.

4.2.2 Local Operating System

Components typically found in local operating systems are also simulated. These include the dispatcher and the device drivers. The local operating systems are multitasking systems with each node capable of utilizing a different time slice. User processes are serviced in a first come first served manner and can be interrupted for any of the following reasons: 1) a control process needs to execute (user process is delayed until the control process releases the processor), 2) the user process exhausts its time slice (user process is placed at the end of the READY QUEUE), 3) the user process attempts to send or receive a message (user process is placed on the MESSAGE BLOCKED QUEUE), or 4) the user process terminates.

The processes serviced by the simulator are capable of performing the following actions: compute, send a message, receive a message, or terminate. A process can access a file by communicating with a FILE PROCESS which is

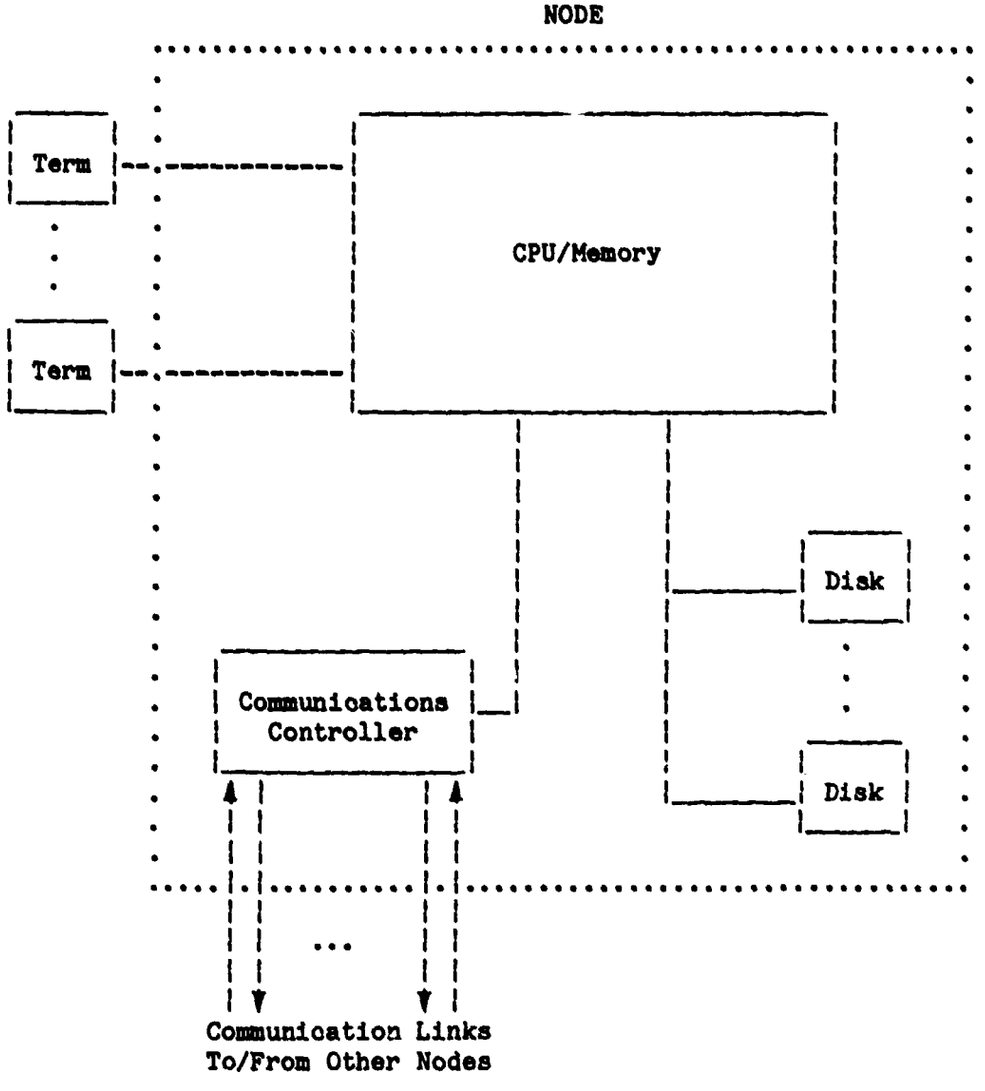


Figure 2. The Architecture Supported by the Simulator for Each Node

activated for the specific purpose of providing access to the file for this process. FILE PROCESSes are the only processes that initiate any disk activity. As far as a user process is concerned, a file access is simply a communication with another process.

The following process queues are maintained: READY QUEUE, DISK WAITING QUEUE, and MESSAGE BLOCKED QUEUE. (See Figure 3.) A newly activated process is placed in the READY QUEUE. The DISPATCHER selects a process from the READY

QUEUE to run on the CPU. If the running process exhausts its time slice, it is returned to the READY QUEUE. If it either attempts to send or receive a message, it is placed in the MESSAGE BLOCKED QUEUE where it remains until either the message is placed in the proper link queue (send operation) or a message is received (receive operation). After leaving the MESSAGE BLOCKED QUEUE, a process returns to the READY QUEUE.

The only processes capable of performing disk input/output on the simulator are FILE PROCESSes. These are executive control processes that are assigned to provide access to the files of the file system. When a file process attempts a disk access, it is blocked and placed in the DISK WAITING QUEUE for processes waiting to access that same disk. As the disk requests are satisfied, these processes are returned to the READY QUEUE.

4.2.3 Message System

The communication system consists of a series of half-duplex connections between pairs of nodes. Messages are transmitted using a store-and-forward method. Messages received at intermediate nodes in a path are stored and forwarded to the next node at a time dictated by the communication policy being utilized. For example, the policy may require that the new message be placed at the end of the queue of all messages to be transmitted on a particular link. (This is the policy utilized in all experiments.)

The message queues available on each node are depicted in Figure 4. If a newly created message is an intranode message, it is placed in the MESSAGE QUEUE; otherwise, it is placed in the LINK QUEUE that corresponds to the communication link over which the message is to be transmitted. Messages are removed from the LINK QUEUEs and transmitted as the communication links become available.

Messages in the MESSAGE QUEUE originate either from processes sending intranode messages or from the communication links connected to the node. Messages destined for processes on the same node as the MESSAGE QUEUE are placed in the appropriate PORT QUEUE of the process to which they are addressed. Messages that have not yet reached their destination are placed in the LINK QUEUE corresponding to the communication link over which the message is to be transmitted.

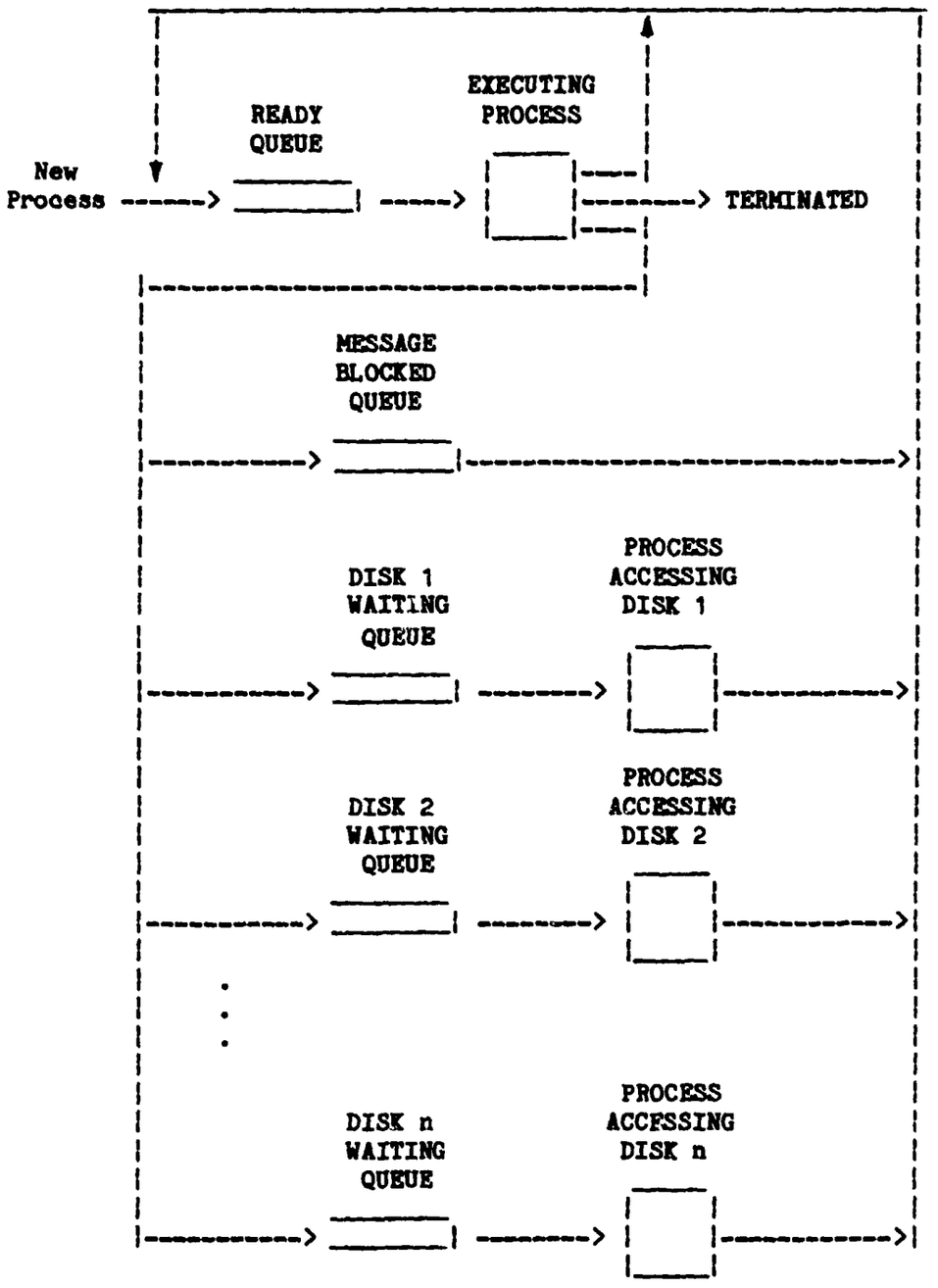


Figure 3. Process Queues on Each Node

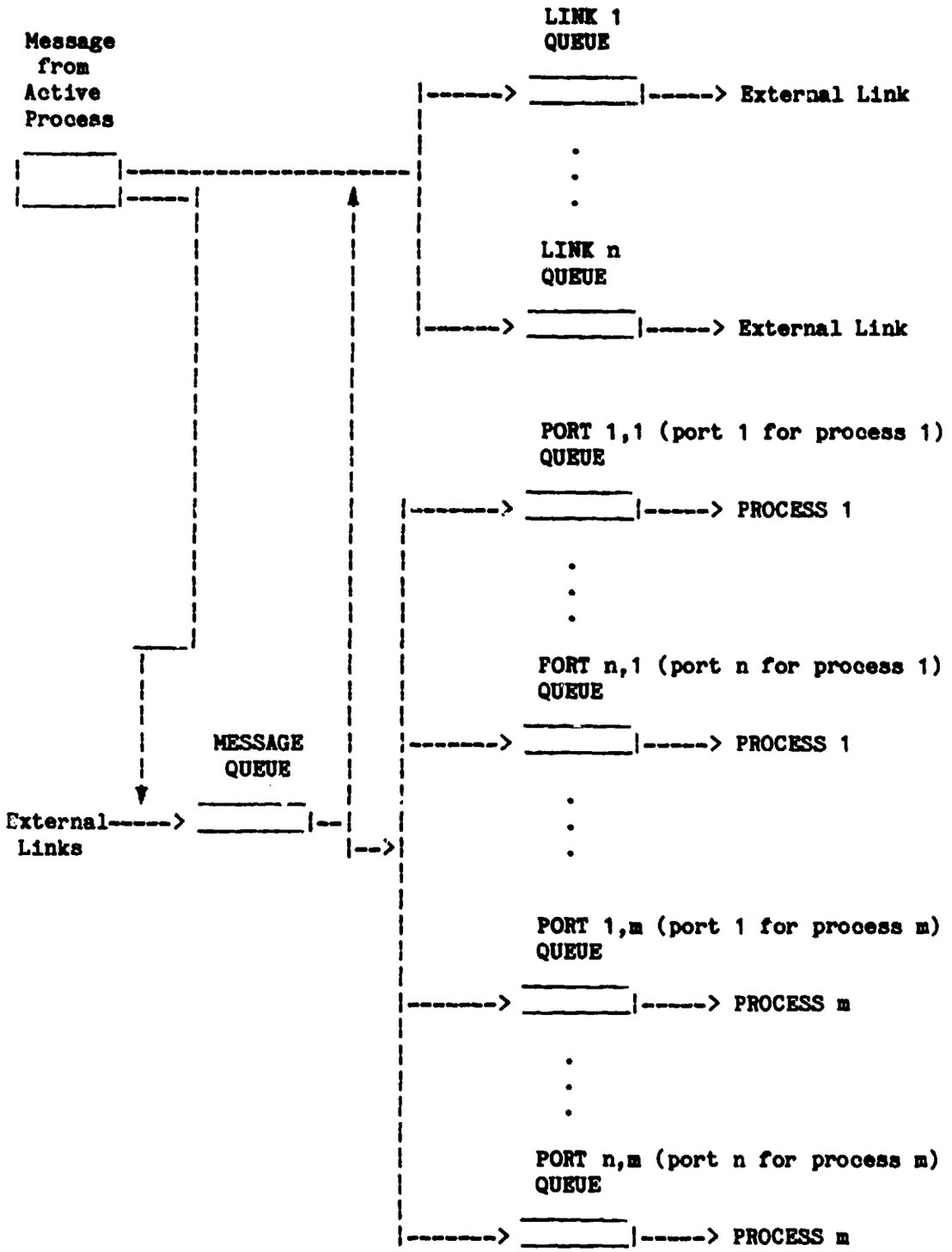


Figure 4. Message QUEUEs on Each Node

4.2.4 Input for the Simulator

The simulator requires the following six types of input:

1. Control model
2. Network configuration (i.e., nodes and their connectivity)
3. Work requests
4. Command files
5. Object files
6. Data files

The nature of these inputs and how they are provided to the simulator is described below.

4.2.4.1 Control Model

There are two possible approaches for representing the control model in the simulator: 1) data to be interpreted by the simulator and 2) code that is actually part of the simulator. The first technique requires that the simulator contain or include a rather sophisticated interpreter in order to provide a convenient language with which one can express a control model that addresses the control problems to a sufficiently low level of detail. The second technique requires the careful construction of the simulator such that those portions of the simulator that express the control model are easily identified and can be removed and modified with minimal effort. The second technique also requires a recompilation of the simulator code each time a control model modification is performed.

The problems involved in constructing a sophisticated interpreter are much greater than those faced in organizing the simulator so that the portions of code expressing the control model are easily isolated. Therefore, in this simulator, the control models are expressed in Pascal and are actually part of the simulator rather than being separate input to the simulator.

4.2.4.2 Network Configuration

The attributes provided as input to the simulator which are concerned with the physical configuration of the FDPS are provided in Table 2. Figure 5 describes the syntax of the statements used to enter the FDPS configuration information. Two types of input can be provided, node configuration information and communication linkage information. Each statement beginning with the letter 'n' describes the configuration of the node which is identified by the digit following the 'n'. This statement describes certain characteristics concerning the processor at the node (memory capacity, processing speed, and

the length of a user time slice) and the peripheral devices (user terminals and disks) attached to the processor. Each statement beginning with the letter 'l' describes a half-duplex communication link between two nodes. It identifies the source and destination nodes by their identification number (the digit following the letter 'n' on statements describing nodes) and indicates the effective bandwidth of the communication link. It is assumed that all messages are transmitted at this speed, and no attempt is made to simulate errors in transmission and the resulting retransmissions.

Table 2. Physical Configuration Input to the Simulator

Node Information

Memory Capacity (bytes)
Processing Speed (Instructions/sec)
Size of a Time Slice (microseconds)
Number of Attached User Terminals
Number of Attached Disks
Disk Transfer Speed (bytes/second)
Average Disk Latency (microseconds)

Link Information

Identities of the Source and Destination Nodes
Bandwidth (bytes/second)

4.2.4.3 Work Requests

Work requests are assumed to originate from two sources: 1) directly from a user, or 2) through command files. The syntax of a work request is given in Figure 6. This syntax is a subset of the command language available through the Advanced Command Interpreter of the Georgia Tech Software Tools System [Akin80].

A work request is basically a specification of a logical network of tasks. The nodes of the logical network represent tasks and the links represent communication paths between the tasks. A node specification includes the following: an optional label to identify the node, a command name (this may name either an object file or a command file), and any I/O redirection. A node can be identified either by its label, if it possesses one, or by its position on the command line. For example, in the command

```
<entry> ::= <link> | <node>

<link> ::= l <from> <to> <bandwidth>  (all links are half-duplex)

<node> ::= n <node id> <memory> <speed> <timeslice> <terminals>
           <disk> <disk speed> <disk latency>

<from> ::= <node id>

<to> ::= <node id>

<node id> ::= <integer>

<bandwidth> ::= <integer (link bandwidth in bytes per second)>

<memory> ::= <integer (main memory in bytes)>

<speed> ::= <integer (average speed of the CPU in instructions per second)>

<timeslice> ::= <integer (microseconds)>

<terminals> ::= <integer (number of attached user terminals)>

<disk> ::= <integer (number of attached disks)>

<disk speed> ::= <integer (transfer speed of disk in bytes/sec)>

<disk latency> ::= <integer (average disk latency in microseconds)>

<integer> ::= <digit> { <digit> }
```

Examples:

```
n 1 256000 5000000 1000 50 3 500000 100
   (Node #1 has 250K bytes of memory, processes at the rate of
    5 MIPS, has a time slice of 1000 microseconds, has 50 user
    terminals attached to it, has 3 disks attached to it,
    each disk can transfer at the rate of 500 000 bytes/sec,
    and each disk has an average latency of 100 microseconds.)

l 5 6 4000000
   (This link connects node 5 to node 6 with a half-duplex
    communication path that can transmit at the rate of
    4 million bytes/sec.)
```

Figure 5. Syntax of FDPS Configuration Input for the Simulator

```
<work request> ::= <logical net>

<logical net> ::= <logical node> { <node separator>
                  { <node separator> } <logical node> }

<node separator> ::= , | <pipe connection>

<pipe connection> ::= [ <port> ] '|' [ <logical node number> ]
                      [ .<port> ]

<port> ::= <integer>

<logical node number> ::= <integer> | $ | <label>

<logical node> ::= [ :<label> ] <simple node>

<simple node> ::= { <i/o redirector> } <command name>
                  { <i/o redirector> }

<i/o redirector> ::= <file name> '>' [ <port> ]  |
                     [ <port> ] '>' <file name>  |
                     [ <port> ] '>>' <file name> |
                                '>>' [ <port> ]

<command name> ::= <command file name> | <object file name>

<label> ::= <identifier>

<file name> ::= <data file name>

<identifier> ::= <letter> { <letter> | <digit> }

<integer> ::= <digit> { <digit> }
```

Examples:

pgm1 | pgm2 1|a 2|b :a pgm3 | pgm4 |c.1 :b pgm5 | pgm6 |.2 :c pgm7

(For an explanation of this example see Figure 7.)

Figure 6. Work Request Syntax
(Based on [AKIN80])

below, the second node has the label 'a' and the command name 'cmnd2'.

```
cmnd1 | :a cmnd2
```

This node can be identified either by the label 'a' or its position '2' but not by its name, 'cmnd2'.

I/O redirection is used to connect ports of task to files in the file system. (The default for I/O is "standard input/output," i.e., the user's terminal.) In the example below, input port number three is connected to file 'in' and output port number one is connected to file 'out'.

```
in>3 cmnd 1>out
```

The specification of the port number in the I/O redirector is optional. If it is omitted, the next unused port number is assumed. Therefore, in the example below, output port number one is connected to file 'out1', output port number two is connected to file 'out2', and output port number three is connected to file 'out3'.

```
cmnd >out1 2>out2 :out3
```

Nodes are separated by node separators which can be either the comma symbol or the vertical bar symbol. The comma symbol is used to separate a node that does not have any output ports connected to any other nodes. The vertical bar symbol or pipe symbol is used to identify the connection of an output port of the node immediately preceding the pipe symbol and the input port of another node. The port numbers and logical node number of the pipe specification may be omitted and default values assumed. If a port number is omitted, the next unused port number for the node possessing the port is used. The logical node number of the pipe specification identifies a node of the logical network. It may either be an integer identifying the position of the node on the command line, the symbol '$' which identifies the last node on the command line, or a node label. If no other node is specified, the node immediately following the pipe symbol is assumed to be the destination of the output of the pipe.

An example of a work request utilizing this syntax is shown in Figure 7. This command consists of seven logical nodes connected in the manner depicted in the figure. It demonstrates several forms of pipe specifications including the use of labels in identifying nodes.

Work Request:

```
pgm1 | pgm2 1|a 2|b :a pgm3 | pgm4 |c.1 :b pgm5 | pgm6 |.2 :c pgm7
    (0)     (1) (2) (3)    (4)    (5)   (6)    (7)    (8)  (9)
```

(0) Output port 1 of pgm1 is connected to input port 1 of pgm2.
(1) Output port 1 of pgm2 is connected to input port 1 of the logical node labeled "a," pgm3.
(2) Output port 2 of pgm2 is connected to input port 1 of the logical node labeled "b," pgm5.
(3) Label for the logical node containing pgm3 as its execution module.
(4) Output port 1 of pgm3 is connected to input port 1 of pgm4.
(5) Output port 1 of pgm4 is connected to input port 1 of the logical node labeled "c," pgm7.
(6) Label for the logical node containing pgm5 as its execution module.
(7) Output port 1 of pgm5 is connected to input port 1 of pgm6.
(8) Output port 1 of pgm6 is connected to input port 2 of pgm7.
(9) Label for the logical node containing pgm7 as its execution module.

Data Flow Graph of the Work Request:

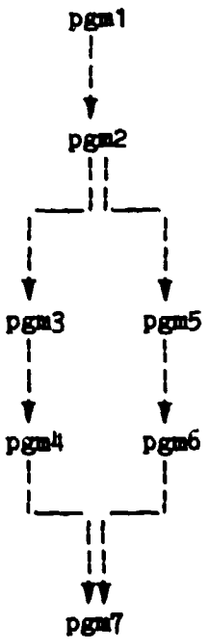


Figure 7. Example of a Work Request

In order to simulate the load generated by users entering work requests from user terminals, a population of work requests is created. The form of the input for creating the work request population is provided in Figure 8. Each line of input contains a series of node identifiers followed by a colon which is followed by a work request. The node identifiers indicate which nodes are to contain the given work request as a member of the node's population of work requests. Therefore, the result of this input is the construction of a population of work requests for each node. In a subsequent paragraph, the nature of the load generator is discussed and indicates how this information is utilized.

<work request population> ::= <work request entry>
.
.
.
<work request entry>

<work request entry> ::= { <node identifier> } : <work request>

<node identifier> ::= <integer>

<work request> ::= (see Figure 6)

<integer> ::= <digit> { <digit> }

Examples:

1 2 3 4 5 : pgm1 | pgm2	{ the work request 'pgm1 | pgm2' is available on nodes 1, 2, 3, 4, and 5 }
1 3 : pgm1	{ the work request 'pgm1' is available on nodes 1 and 3 }

Figure 8. Syntax of Work Request Population Input to the Simulator

4.2.4.4 Command Files

Command files are constructed for the simulator using the syntax described in Figure 9. This input specifies a unique name for the file, the simulated node at which the file resides, and the commands contained in the file. These commands conform to the syntax of work requests presented in

Figure 6. These statements provide one with the ability of constructing command files on particular nodes which are referenced either by commands originating from user terminals or other command files.

```
<command file> ::= C <node id> <command file name>
                   { <work request> }
                   ENDC

<node id> ::= <integer>

<command file name> ::= <up to 8 characters>

<work request> ::= (see Figure 6)

<integer> ::= <digit> { <digit> }
```

Examples:

```
C 1 cfile1
pgm1 | pgm2 1|a 2|b :a pgm3 | pgm4 |c.1 :b pgm5 | pgm6 |.2 :c pgm7
pgm1 | pgm5
ENDC
```

Figure 9. Syntax of Command File Input to the Simulator

4.2.4.5 Object Files

Figure 10 depicts the syntax used to express object files in the simulator. The input specifies a unique name for the file, the simulated node at which the file resides, the length of the file in bytes, and the simulation script. The script contains a series of statements that describe the process actions that are to be simulated. There are five actions which can be simulated: 1) compute, 2) receive a message, 3) send a message, 4) loop back to a previous command a specific number of times, and 5) terminate the process simulation. By appropriately combining these commands, one can construct a script which simulates the activities of a given user process.

4.2.4.6 Data Files

Data files, depicted in Figure 11, are the final type of file which can be presented to the simulator. The data file input contains an identifying

```
<object file> ::= 0 <node id> <object file name> <object file length>
                  { <action> }
                  ENDO

<node id> ::= <integer>

<object file name> ::= <up to 8 characters>

<object file length> ::= <integer>

<action> ::= <comp> | <loop> | <rcv> | <send> |  <term>

<comp> ::= c <# of instructions>

<loop> ::= l <instruction #> <count>

<rcv> ::= r <port>

<send> ::= s <port> <size (bytes)>

<term> ::= t

<# of instructions>, <instruction #>, <count>, <port>,
   <size> ::= <integer>

<integer> ::= <digit> { <digit> }


Examples:

   0 1 object1 1000  (object file is 1000 bytes long)
   c 25              (simulate 25 computation instructions)
   l 1 10            (loop back to the first instruction 10 times)
   r 2               (read a message from port 2)
   s 4 100           (send a message of 100 bytes in length to port 4)
   t                 (terminate the execution of this process)
   ENDO
```

Figure 10. Syntax of Object File Input to the Simulator

name, a node identification indicating the file's simulated location, and a specification of the file size. Data is not actually stored by the simulator.

4.2.5 The Simulator Design

The simulator is composed of several modules. In each module, closely related data structures and the procedures that modify these data structures are defined. The only access to the data structure is through these

<data file> ::= D <node id> <data file name> <size>

<node id> ::= <integer>

<data file name> ::= <up to 8 characters>

<size> ::= <integer (bytes)>

<integer> ::= <digit> { <digit> }

Examples:

D 3 testfile 100000 (defines a data file named 'testfile' which will reside on node 3 and will contain 100,000 bytes of information)

Figure 11. Syntax of Data File Input to the Simulator

procedures. This design allows one to isolate the portion of the simulator that represents the model of control and conduct experiments with various perturbations of the control model. Without this type of design, each perturbation could easily require significant changes to the entire simulator. The major modules of the simulator are described below.

4.2.5.1 Node Module

The NODE MODULE simulates the hardware activities of each node (e.g., the processor and attached disks). This includes the simulation of user activities as specified by process scripts and the simulation of disk traffic. In addition, this module provides the local operating system functions of dispatching, blocking processes for message transmission or reception, and unblocking processes.

4.2.5.2 Message System

All activities dealing with messages are handled by the MESSAGE SYSTEM. Among the services provided by this module are the following: 1) routing of messages, 2) placement of messages in LINK QUEUEs, 3) transmission of messages across a link, 4) transmission of acknowledgement signals to the source end of a link, and 5) placement of messages in PORT QUEUEs.

4.2.5.3 File System

The FILE SYSTEM stores the various types of files, which include object, command, and data files. It stores the scripts for object files and provides access to the scripts. Similarly for command files, it stores the work requests for each command file and controls access to the file. It maintains directories that provide location information and access control information. All executive control actions pertaining to the file system are contained in this module.

4.2.5.4 Command Interpreter

The COMMAND INTERPRETER parses work requests and constructs the task graph describing the initial resource requirements for a work request.

4.2.5.5 Task Set and Process Manager

The TASK SET AND PROCESS MANAGER performs all control activities required to manage all phases of execution of a work request. This includes activating the COMMAND INTERPRETER; communicating with the FILE SYSTEM in order to gather information, allocate files, or deallocate files; perform work distribution and resource allocation; and manage active processes.

4.2.5.6 Load Generator

Work request traffic originating from the user terminals attached to each node is created by the LOAD GENERATOR. A series of work requests provided by a user at a terminal is called a user session. To simulate a user session, the LOAD GENERATOR randomly chooses a session length from a user specified interval. A session starting time (measured in seconds) is also chosen at random from a user specified interval. Each work request for the user session is chosen at random from the population of work requests originally created for each node via the input statements described above (see Figure 8). The LOAD GENERATOR also simulates the "think time" between work requests by randomly choosing a time (measured in seconds) from a user specified interval.

4.2.6 Performance Measurements

Performance measurements are made concerning three types of data: 1) the quantity of message traffic, 2) the magnitudes of various queue lengths and their associated waiting times, and 3) the size of average work request response times and throughput.

To identify the impact of the executive control on the communication system, various communication measurements are obtained. A cumulative total of the number of user messages and control messages over the entire system is maintained. This allows one to compare the number of control messages to the number of user messages and thus identify how the communication system is being utilized. In addition, a count, again categorized by user messages and control messages, is maintained in matrix form to identify the total number of messages originating at a particular node and destined for every other node. Traffic counts on each communication link are also recorded according to their classification as user messages or control messages. Finally, activity in the LINK QUEUEs, where messages wait to be transmitted over each link, is maintained. These measurements include minimum queue length, maximum queue length, average queue length, minimum waiting time in the queue, maximum waiting time, and average waiting time.

In addition to measurements concerned with the LINK QUEUEs, a similar analysis of process queues is performed. The queues on each node that are analyzed are the READY QUEUE (processes waiting for access to the CPU), MESSAGE BLOCKED QUEUE (processes that are either waiting to place a message in a LINK QUEUE or processes waiting to receive a message), and DISK WAITING QUEUEs (processes waiting for access to a particular disk). The types of measurements obtained are identical to those for the LINK QUEUEs.

To identify the effectiveness of the control strategy, measurements are obtained that identify how effectively user processing is accomplished. For each node and cumulatively for all nodes, the following measurements are obtained for user sessions, work requests, and processes:

1. The total number of user sessions, work requests, and processes.
2. The average service time for each user session, work request, and process.
3. The average response time for each user session, work request, and process.
4. The throughput for user sessions, work requests, and processes.

SECTION 5

THE SIMULATION EXPERIMENTS

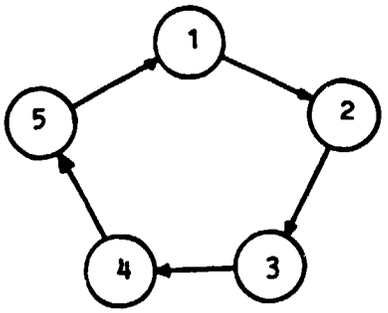
In the second phase of experimentation two groups of simulation experiments designed to measure the performance of the various models in an FDPS environment are conducted. In addition, a number of experiments are conducted with a single node network. In the first group of FDPS experiments, only one work request is processed by the entire network. The intent of this set of experiments is to determine the minimum delay experienced by a work request with each model of control. In the second group of experiments, a load is placed on all nodes. These studies are designed to examine the behavior of the various models of decentralized control operating in a production mode with various physical interconnection topologies. The single node experiments provide a means of comparing the performance of an FDPS to that of isolated uniprocessors.

5.1 THE SIMULATION ENVIRONMENTS

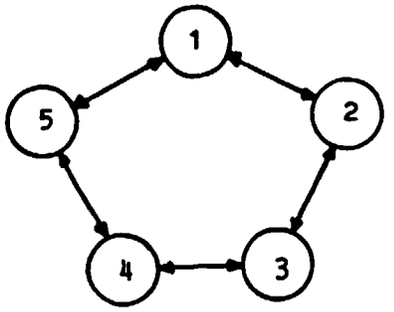
The environment in all FDPS experiments consists of a network of five nodes interconnected in various ways providing five different interconnection topologies: 1) a unidirectional ring, 2) a bidirectional ring, 3) a star, 4) a fully connected network, and 5) a tree. (See Figure 12.) The nodes of each network (see Figure 2) are all homogeneous, and each consists of a processor capable of executing one million instructions per second. Connected to each node are ten user terminals and three disk drives. The disks are assumed identical, each with an average latency of 100 microseconds and a transfer rate of 500,000 bytes per second.

5.1.1 Environmental Variables

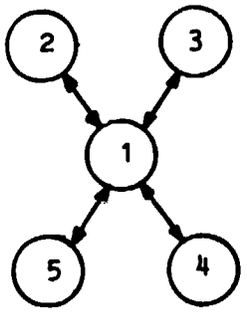
In addition to different topologies, the bandwidth of the communication links and the model of control are also varied for the experiments. Table 3 provides a brief comparison of the various models. Only the first four models of control (XFDPS.1, XFDPS.2, XFDPS.3, and XFDPS.4) are utilized in these initial experiments. Models XFDPS.5 and XFDPS.6 differ from model XFDPS.1 in details that are not examined in these experiments. Therefore, they are not considered in these experiments because their observable results will be identical to those of XFDPS.1. It is instructive, though, to note that not



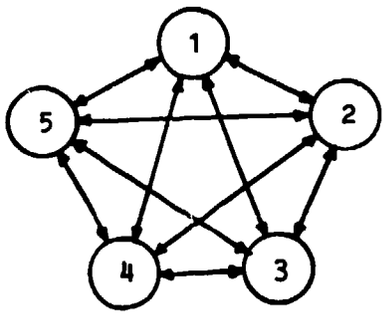
Unidirectional Ring



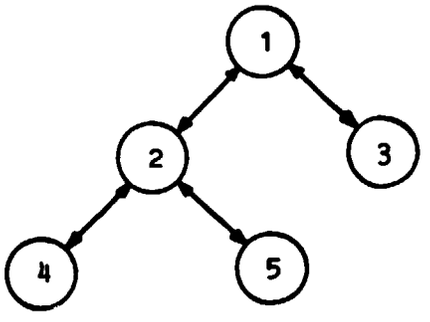
Bidirectional Ring



Star



Fully Connected



Tree

Figure 12. Network Interconnection Topologies

all model variations result in performance differences. Finally, it should be noted that the central directory of model XFDPS.2 is maintained on node 1 in all experiments.

5.1.2 Environmental Constants

Several environmental features remain constant for all experiments. In all cases, it is assumed that all control messages are 50 bytes long. All control models utilize the same policy for distributing work and allocating resources. This policy simply requires all processes to execute on the node where the object code for that process resides. There is only one copy of the object code for each process in the network for these initial experiments. The work distribution and resource allocation policy utilized for these tests requires that data files be accessed at the location where they originally reside and not be moved prior to execution. In every experiment, all files are unique thus leaving the control with only one resource allocation alternative.

The work requests arriving at all nodes are of the type 'in> cmnd'. The data file 'in' provides input to the process resulting from the loading of the object file 'cmnd'. This provides an environment in which files are accessed only by means of reads thus eliminating the possibility that certain work requests are either delayed or aborted due to insufficient resources. Therefore, it is guaranteed that all control activity results in the successful completion of a work request.

In all cases, the object file 'cmnd' and data file 'in' are located on the same node. This means that all file accesses are local file accesses and thus control message traffic is free of competition by user messages for communication resources. This provides an environment in which the effects of the control models can be more directly observed without the influence of an unpredictable collection of user messages.

The object files in each case specify the execution of the same script which is depicted in Figure 13. This script describes a process that alternately computes and reads from a data file for 501 iterations. Given the speed of the processors utilized in the experiments, this results in a CPU utilization of approximately 5 seconds for each process.

Table 3. Comparison of the Control Models

Model	File System Directory	Technique for Gathering Availability Information	Time When Files are Reserved or Locked	How is the Task Graph Maintained
1	partitioned and distributed	query all nodes	before resource allocation and work distribution decision	single structure on node where work request arrived
2	single centralized copy	query the central node	before resource allocation and work distribution decision	single structure on node where work request arrived
3	partitioned and distributed	first query locally and then query globally if necessary	before resource allocation and work distribution decision	single structure on node where work request arrived
4	identical copies replicated on all nodes	all queries are delayed until the control vector arrives	before resource allocation and work distribution decision	single structure on node where work request arrived
5	partitioned and distributed	query all nodes	after resource allocation and work distribution decision	single structure on node where work request arrived
6	partitioned and distributed	query all nodes	before resource allocation and work distribution decision	multiple subgraphs on the nodes involved in the execution of the tasks

```
c  10000      { 10,000 compute instructions }
r  1          { read from port 1 }
l  1  500     { loop back to instruction one 500 times }
t             { terminate the process }
```

Figure 13. The Script Utilized By All Processes

5.2 GROUP 1 EXPERIMENTS

5.2.1 The Environment

The first group of experiments is designed to demonstrate the minimum delay experienced by a single work request as a result of utilizing each model of control. In this set of experiments, all topologies are investigated in addition to various bandwidths ranging from 1200 to 500,000 bytes per second. These experiments examine situations in which work requests arrive at both nodes 1 and 2. In addition, the location of the object-data file pairs named in the work request are varied over all five nodes.

Each of these tests requires the simulator to process only one work request, thus eliminating competition for resources by other work requests. The work request response times for each environment (model, topology, bandwidth, and location of object-data file pair) are provided in Appendix 2.1.

5.2.2 Observations

A comparison of the results of this set of experiments can be seen in Figures 14 and 15. In Figure 14, the results of work requests arriving at node 1 can be seen. Node 1 is chosen in order to demonstrate how XFDPS.2 (the model with a centralized file system directory located on node 1) can benefit from the location of a work request. In all cases, model XFDPS.2 provides the smallest response times. When the work request arrives at another node (e.g., node 2) XFDPS.2 no longer provides the minimum response time in all cases.

The sensitivity of XFDPS.2 to the location of the work request can be attributed to the location of the central file system directory on node 1. If a work request arrives at node 1, all resource allocation can be performed without requiring the transmission of any control messages. The only control messages needed are those necessary to activate the file processes for each file named in the work request. These messages are transmitted once the files

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 > 2 = 3	4 > 1 = 3 > 2
50,000	4 > 1 > 2 = 3	4 > 1 = 3 > 2
100,000	4 > 1 > 2 = 3	4 > 1 = 3 > 2
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 > 2 = 3	4 > 1 = 3 > 2
50,000	4 > 1 > 2 = 3	4 > 1 = 3 > 2
100,000	4 > 1 > 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

STAR

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 > 2 = 3	4 > 1 = 3 > 2
50,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 > 2 = 3	4 > 1 = 3 > 2
50,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

TREE

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 > 2 = 3	4 > 1 = 3 > 2
50,000	4 > 1 > 2 = 3	4 > 1 = 3 > 2
100,000	4 > 1 > 2 = 3	4 > 1 = 3 > 2
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

Notation: i > j means response time using model i is greater than that using j
i = j means response time using model i is similar to that using j

Figure 14. Comparison of the Response Times for Models 1, 2, 3, and 4 that Were Obtained from the Group 1 Experiments in Which Work Requests Arrived at Node 1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	2 > 4 > 1 > 3	2 > 1 = 3 > 4
50,000	4 > 2 > 1 > 3	4 > 1 = 2 = 3
100,000	4 > 1 = 2 > 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 2 > 1 > 3	4 > 1 = 3 > 2
50,000	4 > 1 = 2 > 3	4 > 1 = 2 = 3
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

STAR

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 1 = 2 > 3	4 > 1 = 3 > 2
50,000	4 > 1 = 2 > 3	4 > 1 = 3 > 2
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 2 > 1 > 3	4 > 2 > 1 = 3
50,000	4 > 2 = 1 > 3	4 > 1 = 2 = 3
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

TREE

Bandwidth (bytes/sec)	Object and Data File at Node 1	Object and Data File at Node 3
1200	4 > 2 > 1 > 3	4 > 2 > 1 = 3
50,000	4 > 1 = 2 > 3	4 > 2 > 1 = 3
100,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3
500,000	4 > 1 = 2 = 3	4 > 1 = 2 = 3

Notation: i > j means response time using model i is greater than that using j
i = j means response time using model i is similar to that using j

Figure 15. Comparison of the Response Times for Models 1, 2, 3, and 4 that Were Obtained from the Group 1 Experiments in Which Work Requests Arrived at Node 2

have been allocated. If the work request arrives at node 2, a message must be sent to node 1 in order to allocate the resources. Once the resources have been allocated, the messages to activate file processes can be sent. Therefore a two stage operation with two sets of messages results from this scenario.

XFDPS.1 and XFDPS.3 provide an alternate strategy which explains their superior performance to XFDPS.2 when the work request arrives at node 2. In these models, file allocation and file process activation are accomplished with one message because the directory for a file and the file itself reside on the same node. Therefore, once a file has been allocated, the file process can be activated with an intranode operation.

In all but two cases, XFDPS.4 results in the largest response time of all the models. Only when the work request arrives at node 2 in a network consisting of a unidirectional ring with a bandwidth of 1200 bytes per second does this model perform better than the other models. This particular topology provides the longest paths between nodes thus making it quite susceptable to communication problems. Model XFDPS.4 performs better at low bandwidths than the other models for this particular topology because only one message is present on the communication net once a work request is being processed. During the resource allocation phase, the update vector (UPV) circulates about the ring; and, after this step, the control vector (CV) is present on the ring. In all other models, multiple messages are utilized to process a work request; thus, at low bandwidths, message throughput becomes a problem.

Finally, the outstanding performance of XFDPS.3 when the object and data files named in a work request reside on the same node as the work request should be noted. This is a clear demonstration of the savings possible with this policy. One should also note that the performance of XFDPS.1 and XFDPS.3 are identical when the named files are on a node different than the one receiving the work request.

5.3 GROUP 2 EXPERIMENTS

The first set of experiments demonstrates fundamental differences in the performance of the models when handling individual work requests, but this

type of experiment can often be deceiving. When multiple work requests are processed concurrently, the simultaneous demands on resources can result in unexpected delays which cannot be anticipated with the data obtained from the first set of experiments.

5.3.1 The Environment

The goal of this set of experiments is to simulate and examine a production environment. It would be desirable to establish identical loads for all experiments, but the nature of the problem makes this impossible. The basic environment consists of a network of five nodes with ten user terminals attached to each node. To provide an identical load, one would have to guarantee that the work requests will be presented to the simulator in the same order for each experiment. The control models, though, are composed of autonomous components and by their design will process work requests on each node at different rates as demonstrated by the results of the group 1 experiments. This implies that even if the work requests at each node are presented in the same order, the load provided to the simulator will be different because the timing of work request arrivals may vary.

To clarify this point, consider the following example. Assume the loads provided to nodes 1 and 2 are as shown in Figure 16. This figure depicts the order in which the work requests arrive at each node. Because the control models process work requests at different rates, different processing sequences are obtained for the control models. Figure 17 depicts the sequence for model 1 and Figure 18 depicts that for model 2. Thus, although the loads at each node are controlled, it is impossible to control the sequence of work requests on all nodes collectively.

Load at Node 1	Load at Node 2
WR1	WR5
WR2	WR6
WR3	WR7
WR4	WR8

Figure 16. Example of Loads Presented to Two Nodes

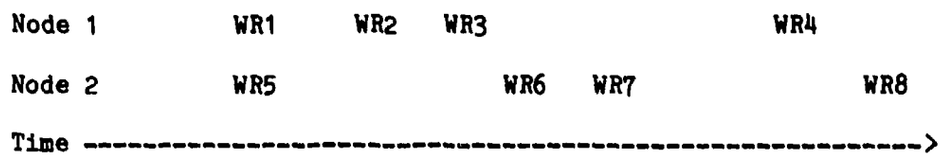


Figure 17. Sequence of Work Request Arrivals When Using Model 1

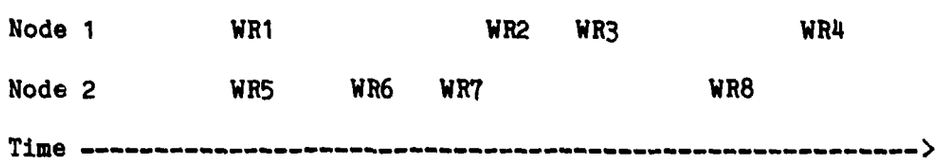


Figure 18. Sequence of Work Request Arrivals When Using Model 2

Since identical loads cannot be provided, we attempt to construct an unbiased load. Each terminal issues its first work request at a time measured in seconds corresponding to an integral value chosen at random from the interval [1, 15]. After a work request has completed, the arrival time (measured in seconds) of the next work request from the terminal is again chosen by selecting a random value in the interval [1, 15] as the delay from the termination of the previous work request. The work requests are chosen at random from a common pool of work requests. Each work request in the pool is of the type described earlier in section 5.1.2 naming object-data file pairs in which both the object file data file reside on the same node. There is an equal number of object-data file pairs on each node. Therefore, the probability that a newly arrived work request names an object-data file pair residing on node i is 1/5 for i = 1, 5.

In order to obtain steady state data, the taking of measurements is delayed until a simulation time of 30 seconds after the start of the test. This insures that all terminals are active and are into their normal activities. Measurements are then taken until 330 seconds into the simulation thus providing a measurement interval of 5 minutes. This provides observation of the processing of over 200 work requests. Longer simulation intervals, though desirable, are not practical due to the extensive computation necessary

to simulate the level of detail provided by the control models being examined. It has been observed for most runs that over three hours of computing time on a Prime 550 are required. (The performance of the Prime 550 is approximately 80% of that of an IBM 370/158 and 35% of that of a VAX 11/780 [Henk81].) Over 160 simulation runs have been made during the process of this research.

In this set of experiments, the following three factors are varied: 1) control model, 2) topology, and 3) bandwidth. Experiments utilizing all possible combinations of these factors are run. The results of these experiments are provided in Appendix 2.2.

5.3.2 Observations

The most distinguishing feature of the results of these tests is the lack of significant variation in average response time for experiments utilizing all models and topologies with bandwidths 1200 bytes per second or larger. In all cases, the LINK QUEUEs have an average length of between one and two messages, implying that the communication system does not prove to be a bottleneck.

To demonstrate that the values for average response time could be explained by delays due to the intranode multitasking of processes, experiments utilizing the extremely high bandwidth of 2.5 million bytes per second are conducted. The results are very similar to those obtained with much lower transmission rates. In addition, a simulation of a single node network is conducted. This also results in an average response time not significantly different. (The results of the single node simulation are provided in Appendix 2.3.)

In most cases when the bandwidth is lowered to values below 600 bytes per second, a statistically significant increase in response time is observed. In most cases, either XFDPS.2 or XFDPS.4 provided the smallest average response time values. It is necessary, though, to reduce the bandwidth to extremely low values in order to observe these differences, thus leading us to conclude that as far as constrasting the various models is concerned, the data is rather inconclusive.

Finally, the results of the experiments with model XFDPS.2 provide one further observation. Recall that in this model a single centralized file system directory is maintained. All file system requests are handled by the

node housing this directory. Therefore, one would expect the performance of this node to be somewhat degraded due to the control activity required to satisfy the file system requests. The results, though, show that this is not the case. The average response times for work requests arriving at the node where the central directory is maintained (node 1) do not differ significantly from those on other nodes. This result implies that the amount of file system management work is rather negligible, thus, it does not lead to any performance degradation.

5.4 SINGLE NODE NETWORK EXPERIMENTS

5.4.1 The Environment

This set of experiments is considered separately from those described above because its purpose is not to analyze the relative performance of the control models. These experiments are designed to provide a standard upon which the other results can be compared in order to determine the impact of distributed processing on average response time for work requests.

The configuration of the single node comprising the network in this set of experiments is identical to that for each node in the other experiments. The work requests name object-data file pairs and the script for the object file is the same as that employed in the first two groups of experiments. Since there is no internode communication, the choice of the control model is of no consequence, and therefore XFDPS.1 is arbitrarily selected.

5.4.2 Observations

Five simulations are conducted and the results of those runs are presented in Appendix 2.3. The values for average response time from these experiments are similar to those found in the first group of experiments when bandwidths greater than 600 bytes/sec are used.

SECTION 6

CONCLUSIONS

6.1 QUALITATIVE ASPECTS OF THE MODELS

The evaluation of the control models would be incomplete if consideration were given only to the quantitative results provided by the simulation experiments. It is also important to examine certain qualitative aspects of the models which were not quantitatively evaluated. These aspects include the ability to provide fault-tolerant operation (e.g., graceful degradation and restoration), the ability for the system to expand gracefully, and the ability to balance the system load.

6.1.1 XFDPS.1

The XFDPS.1 model is a truly distributed and decentralized model of control. In this model, resources are partitioned along node boundaries and managed by components residing on the same node as the resource. This design enables the system to remain in operation in the presence of a failure. In such a situation, those nodes not available are simply not contacted when queries concerning resources are made. The failed nodes are also not considered as locations for the execution of tasks during the formulation of the work distribution and resource allocation decision.

This model of control requires some activity on the part of all nodes in order to satisfy each work request. There is no single node that is by design supposed to receive any more activity than any other node; instead, the work is spread across all nodes. In addition, global information for the work distribution and resource allocation decision is obtained for each work request as it is processed. This global data enables the control to better balance the load across the network.

This control model is not without its problems. The global searches for resources that occur for every work request may be unnecessary (e.g., in those instances in which only local resources are required). Short local jobs therefore suffer at the expense of the longer jobs utilizing non-local resources.

6.1.2 XFDPS.2

XFDPS.2 utilizes a single centralized file system directory. On the surface, this model appears to be simple to implement. A central directory is maintained, and all file system queries are sent to the node housing that directory. However, problems result when fault-tolerant operation is desired. No longer can a single central directory be maintained because the loss of the node housing the directory would be catastrophic. Alternative strategies which provide for fault-tolerant operation (see for example Garcia-Molina's technique described in [Garc79] for providing fault tolerance in a centralized locking distributed data base system) significantly complicate the design of the control as well as require a significant expenditure of resources in order to recover from a failure. It should be noted that the simulation of XFDPS.2 does not account for the overhead required to provide fault-tolerant operation. Therefore, the average work request response times observed in the experiments are lower than would be expected if the necessary control features for providing fault-tolerant operation were present.

Model XFPDS.2 also has problems with growth. When a new node is introduced into the system, a large amount of work is required to update the central directory to add the resources of the new node. This factor can be quantified and will be the subject of future experiments.

6.1.3 XFDPS.3

The XFDPS.3 model is similar to XFDPS.1. It differs in its policy for obtaining file availability information. First a local search is made. If all resources are found, they are utilized; otherwise, a global search for resources is conducted. As described in Section 5, this model provides faster response to work requests utilizing only local resources as expected. Due to its information gathering policy, the potential for utilizing distant resources in order to balance the load is sacrificed because resource availability on other nodes may never be considered.

6.1.4 XFDPS.4

XFDPS.4 utilizes redundant copies of the file system directory on all nodes. Access to the directory is restricted to the node possessing the control vector that is passed among the nodes of the network. This model tends to work somewhat like a batch system by delaying file system requests

until the control vector (CV) is received and processing these requests as a batch.

The presence of the replicated file directory implies that there is both duplication of information storage and duplication of effort as consistency is maintained across the replicated copies. Since file system requests are delayed until the CV arrives, jobs with very short service times may experience unusually large response times. Finally, as with XFDPS.2, the introduction of a new node requires a large amount of work in order to update the replicated directories.

6.1.5 XFDPS.5

XFDPS.5 is nearly identical to XFDPS.1, differing only in its policy of not locking or in any way reserving resources prior to the formulation of a work distribution and resource allocation decision. With this policy, resources are not expected to be needlessly tied up in most cases. A problem does exist if the chosen resources cannot be locked once selected for allocation. In this case, a new resource allocation decision must be made and already allocated and locked resources may need to be released.

6.1.6 XFDPS.6

XFDPS.6 differs from XFDPS.1 in the manner in which the task graph and task activation are handled. In this model, the tasks of a work request that are chosen to execute on the same node are presented to the PROCESS MANAGER of the selected node collectively. A task graph identifying this collection of tasks is constructed and task activation and termination are handled by the PROCESS MANAGER. Thus, the TASK SET MANAGER need send only one message to each of the nodes utilized by the work request in order to activate all tasks. In addition, only one termination message is received from each node. Further savings are provided because the PROCESS MANAGER on the node where the tasks are executing can immediately release the resources utilized by the tasks as each task terminates.

6.2 CONCLUSIONS

One must remember when analyzing the results in Appendix 2 that only control message traffic is present during these simulation experiments. The simulation experiments may be inconclusive in establishing the relative merits

of the various models. They do, though, demonstrate the utility of the fully distributed processing concept. Even networks with communication links possessing low bandwidths appear to be feasible candidates for fully distributed processing if the message traffic is held mainly to control messages. In particular, the experiment with the single node network leads one to expect that there will be little or no performance loss experienced with an FDPS.

One of the most important results of this research is the production of a simulator for the analysis of fully distributed processing systems. The experience gained from the simulator has been the basis for the proposal of several interesting experiments to be conducted in the future.

SECTION 7

FUTURE EXPERIMENTS

This work has suggested several future experiments. First, networks of increasing numbers of nodes, possibly 10, 15, and 20 node networks, will be investigated to determine at what point the utility of the various models is lost. In addition, experiments with both user message traffic and control message traffic will be investigated in order to observe the sensitivity of the various models in the presence of a busy communication system. Different resource allocation and work distribution algorithms will be instrumented into the simulator in order to determine under what conditions each algorithm is appropriate.

The issue of the dynamic addition and deletion of resources will also be examined. This will demonstrate how gracefully the various models can adapt to a growing system. These experiments will also examine the fault-tolerant capabilities of the various models.

REFERENCES

[Akin80] Akin, T. Allen, Flinn, Perry B., and Forsyth, Daniel H., "User's Guide for the Software Tools Subsystem Command Interpreter (The Shell)," School of Information and Computer Science, Georgia Institute of Technology, 1980.

[Davi79] Davies, D. W., Barber, D. L. A., Price, W. L., and Solomonides, C. M., Computer Networks and Their Protocols, John Wiley and Sons, 1979.

[Ensl78] Enslow, Philip H., Jr., "What is a 'Distributed' Data Processing System?" Computer (January, 1978): 13-21.

[Ensl81] Enslow, Philip H., Jr., and Saponas, Timothy G., "Distributed and Decentralized Control in Fully Distributed Processing Systems - A Survey of Applicable Models," Technical Report No. GIT-ICS-81/02, Georgia Institute of Technology, February, 1981.

[Garc79] Garcia-Molina, H., "Performance Comparison of Update Algorithms for Distributed Databases, Crash Recovery in the Centralized Locking Algorithm," Progress Report No. 7, Stanford University, 1979.

[Henk81] Henkel, T., "Superminis: An Alternative," Computerworld (July 13, 1981): 17-18.

[Jens78] Jensen, E. Douglas, "The Honeywell Experimental Distributed Processor - An Overview," Computer (January, 1978): 28-38.

[Sapo80] Saponas, Timothy G., and Crews, Phillip L., "A Model for Decentralized Control in a Fully Distributed Processing System," COMPCON Fall 80 (September, 1980).

APPENDIX 1

CONTROL MODEL PSEUDO CODE

1.1 PSEUDO CODE FOR THE XFDPS.1 CONTROL MODEL

1.1.1 System initiator

```
 1:  process system_initiator;
 2:  { Every node possesses one of these processes.  This process
 3:    initiates a node in the network by assigning 'task_set_manager'
 4:    processes to each connected user terminal, activating the
 5:    'file_system_manager' process, and activating the
 6:    'processor_utilization_manager' process. }
 7:
 8:  begin
 9:     for every attached user terminal i do
10:        task_set_manager (TERMINAL, i);
11:     endfor;
12:     file_system_manager;
13:     processor_utilization_manager;
14:  end system_initiator;
```

1.1.2 Task Set Manager

```
 1:  process task_set_manager (case input_origin: inp_orig of
 2:                               TERMINAL: (term: terminal_address);
 3:                               CMNDFILE: (fd: filedescriptor)
 4:                            end);
 5:  { Every terminal and every executing command file are assigned
 6:    a 'task_set_manager' process.  When a process of this type
 7:    is activated, one of two sets of parameters is passed to it
 8:    depending upon the source of input to the process.  If the
 9:    process is assigned to handle input from a terminal, the
10:    address of the terminal is provided.  If the process is
11:    assigned to handle input from a command file, the file
12:   descriptor for the command file is provided. }
13:
14:  var
15:     tg: task_graph_pointer;
16:     command_line: string;
17:     msg: message_pointer;
18:
19:  begin
20:     while <either the terminal is attached or the end
21:        of the file has not been reached> do
22:        <get the next work request and store it in command_line>;
23:        new (tg);
24:        parse (command_line, tg);
25:        <send a message of type M1 (file availability request) to
26:           the file_system_manager on this node that contains the
27:           names of files need for this work request>;
```

```
28:          <send a message of type M2 (processor utilization request)
29:             to the processor_utilization_manager on this node>;
30:          <wait for a message from processor_utilization_manager>;
31:          <store processor utilization information in tg^>;
32:          <wait for a message from file_system_manager>;
33:          <store file availability information in tg^>;
34:          if work_distributor_and_resource_allocator (tg) = ERR then
35:             { work distribution and resource allocation
36:             decision could not be made }
37:             <report error>;
38:             if input_origin = CMNDFILE then
39:                exit { leave the loop }
40:             else
41:                next { next iteration of loop }
42:             endif;
43:          endif;
44:          <send a message of type M3 (file lock and release request)
45:             to the file_system_manager on this node>;
46:          <wait for a message from file_system_manager>;
47:          if <all locks could not be applied> then
48:             <report error>;
49:             <send a message of type M4 (file release request)
50:                to the file_system_manager on this node>;
51:             if input_origin = CMNNDFILE then
52:                exit { leave the loop }
53:             else
54:                next { next iteration of loop }
55:             endif;
56:          endif;
57:          for <all files chosen to be copied before execution> do
58:             <send a message of type M5 (file copy request) to the
59:                file_system_manager on this node>;
60:          if <files need copying> then
61:             <wait for a message from the file_system_manager>;
62:          endif;
63:          for <each node i chosen to execute parts of the
64:             work request> do
65:             <send a message of type M6 (process activation request)
66:                to the process_manager on node i>;
67:          endfor;
68:          repeat
69:             <wait for a termination message from a process_manager
70:                or a request to terminate the command file from
71:                the process_manager that activated this
72:                task_set_manager>;
73:             if <this is a termination message from a
74:                process_manager> then
75:                <mark the terminated task as completed in tg^>;
76:                <send a message of type M4 (file release request)
77:                   to the file_system_manager on this node>;
```

```
78:              if <the termination status indicated that the
79:                 process terminated due to an error> then
80:                 for <each node i still running parts of this
81:                    work request> do
82:                    <send a message of type M7 (process kill request)
83:                       to the process_manager on node i>;
84:                 endfor;
85:              endif;
86:           else
87:              for <every task of the work request> do
88:                 if <the task has not completed> then
89:                    <send a message of type M7 (process kill request)
90:                       to the process_manager responsible for
91:                       the task>;
92:                 endif;
93:              endfor;
94:              break; { exit the loop }
95:           endif;
96:        until <all tasks have terminated>;
97:     endwhile;
98:  end task_set_manager;
```

1.1.3 File System Manager

```
 1:  process file_system_manager;
 2:  { Every node possesses one of these processes.  This process
 3:    satisfies various requests concerning the file system.
 4:    This is accomplished by communicating with the file_set_managers
 5:    on all nodes. }
 6:
 7:  var
 8:     msg: message_pointer;
 9:     favptr: file_availability_rec_pointer;
10:     flrprt: file_lock_and_release_rec_pointer;
11:
12:  begin
13:     loop
14:        <wait for a message of any type (let msg point to
15:           the message)>;
16:        case msg^.message_type of
17:           M1: { file availability information request }
18:              begin
19:                 new (favptr);
20:                 <insert the record favptr points to into the
21:                    list of fav_recs>;
22:                 <record the names of the files identified in msg^>;
23:                 for <each node i> do
24:                    <send a message of type M8 (file availability
25:                       request) to the file_set_manager on node i
26:                       that contains the names of all files>;
27:                 endfor;
28:              end;
```

```
29:          M3: { file lock and release request }
30:             begin
31:                new (flrptr);
32:                <insert the record flrptr points to into the
33:                   list of flr_recs>;
34:                for <each node i> do
35:                   <send a message of type M9 (file lock and
36:                      release request) to the file_set_manager
37:                      on node i that contains the names of all
38:                      files from msg^ that are identified
39:                      as being located at node i>;
40:                endfor;
41:             end;
42:          M4: { file release request }
43:             begin
44:                for <each node i> do
45:                   <send a message of type M10 (file release
46:                      request) to the file_set_manager on
47:                      node i that contains the names of all
48:                      files from msg^ that are identified as
49:                      being located at node i>;
50:                endfor;
51:             end;
52:          M5: { file copy request }
53:             begin
54:                new (fmvptr);
55:                <insert the record fmvptr points to into the list
56:                   of fmv_recs>;
57:                for <each file named in msg^> do
58:                   <insert the file name into fmvptr^>;
59:                   <send a message of type M11 (create file request)
60:                      to the file_set_manager on the node where
61:                      the file is to be copied>;
62:                endfor;
63:             end;
64:          M12: { file availability info from file_set_manager }
65:             begin
66:                <let favptr point to the fav_rec that msg^
67:                   is a response to>;
68:                <fill in the availability information in favptr^>;
69:                if <responses from all file_set_managers
70:                   have been received> then
71:                   <send a message of type M16 (file availability
72:                      information) to the task_set_manager
73:                      identified by a field of favptr^>;
74:                endif;
75:             end;
76:          M13: { file lock and release results from file_set_manager }
77:             begin
78:                <let flrptr point to the flr_rec that msg^
79:                   is a response to>;
80:                <fill in the lock and release results in flrptr^>;
```

```
                if <responses from all file_set_managers
                   that were contacted have been received> then
                   <send a message of type M17 (results of file
                      lock and release request) to the task_set_manager
                      identified by a field of flrptr^>;
                endif;
             end;
          M14: { result of file creation request from file_set_manager }
             begin
                { This message is part of a series of messages
                  used to copy a file from one node to another.
                  At this point, file processes have been activated
                  at both the sending and receiving nodes.  The
                  next step is to send a signal to the sending
                  process to begin transmission. }
                <send a message of type M18 (signal to begin copy)
                   to the sending file process in the copy
                   operation>;
             end;
          M15: { copy completion signal from a file process }
             begin
                <let fmvptr point to the fmv_rec that msg^
                   is a response to>;
                <record in fmvptr^ that the copy operation
                   indicated in msg^ has been completed>;
                if <all copy operations have been completed> then
                   <send a message of type M19 (results of file
                      copy request) to the task_set_manager
                      identified by a field of fmvptr^>;
                endif;
             end;
       endcase;
    endloop;
 end file_system_manager;
```

1.1.4 Processor Utilization Manager

```
process processor_utilization_manager;
{ Every node possesses one of these processes.  This process
  records the latest processor utilization information received
  from each node's processor_utilization_monitor; it provides
  task_set_managers with this information on demand; and
  if it does not hear from a processor_utilization_monitor
  within a particular interval of time, it records the processor
  as down and attempts to contact that processor utilization_monitor. }

var
   msg: message_pointer;
   pcutil: array [NODES_OF_THE_NET] of pc_utilization;
```

```
14:  begin
15:     loop
16:        <wait for a message of any type (let msg point to
17:           the message)>;
18:        case msg^.message_type of
19:           M2: { pc utilization information request }
20:              begin
21:                 <send a message of type M20 (pc utilization
22:                    information) to the task_set_manager that
23:                    sent the message and is identified in msg^>;
24:              end;
25:           M3: { pc utilization information from monitor }
26:              begin
27:                 <record information in msg^ in pcutil [msg^.node]>;
28:                 <reset deadman timer for information arriving
29:                    from node msg^.node>;
30:              end;
31:           M22: { deadman timer signal - this indicates that a
32:                  processor_utilization_monitor has not reported
33:                  within the required time }
34:              begin
35:                 pcutil [msg^.node] := NOT_AVAILABLE;
36:                 <send a message of type M23 ("are you alive?"
37:                    query) to the processor_utilization_monitor
38:                    on node msg^.node>;
39:              end;
40:        endcase;
41:     endloop;
42:  end processor_utilization_manager;
```

1.1.5 Processor Utilization Monitor

```
 1:  process processor_utilization_monitor;
 2:  { Every node possesses one of these processes.  This process
 3:    records various performance measurements and computes a
 4:    processor utilization value that is periodically transmitted
 5:    to all processor_utilization_managers. }
 6:
 7:  begin
 8:     loop
 9:        <gather performance measurements>;
10:        <compute processor utilization value>;
11:        for <each node i> do
12:           <send a message of type M21 (processor utilization
13:              information) to the processor_utilization_manager
14:              on node i>;
15:        endfor;
16:        <sleep until it is time to gather more measurements>;
17:        <wait until it is time to gather more measurements
18:           or a message from a processor_utilization_manager
19:           arrives>;
20:     endloop;
21:  end processor_utilization_monitor;
```

1.1.6 Process Manager

```
process process_manager;
{ Every node possesses one of these processes.  This process
  manages the processes that are executing on its node. }

var
   pcbptr: process_control_block_pointer;
   process_name_table: process_name_to_pcbptr_map;
   msg: message_pointer;

begin
   loop
      <wait for the arrival of a message (let msg point
         to the message)>;
      case msg^.message_type of
         M6: { process activation request }
            begin
               if <process type is an object file> then
                  new (pcbptr);
                  <record process identifying information
                     and pcbptr in process_name_table>;
                  <fill in the necessary information in pcbptr^>;
                  <initiate the loading of the process>;
               else
                  task_set_manager (CMNDFILE, msg^.file_descriptor);
                  <record process identifying information
                     and task_set_manager identification in
                     process_name_table>;
               endif;
            end;
         M7: { process kill request }
            begin
               <find the process in process_name_table>;
               if <the process is an object file> then
                  <terminate the process>;
                  <unload the process>;
                  <dispose of the process control block>;
                  <send a message of type M24 (process
                     termination message) to the task_set_manager
                     that activated the process>;
               else { the process is a command file }
                  <send a message of type M25 (request to terminate
                     the execution of a command file) to the
                     task_set_manager executing this command file>;
               endif;
            end;
      endcase;
   endloop;
end process_manager;
```

1.1.7 File Set Manager

```
 1:  process file_set_manager;
 2:  { Every node possesses one of these processes.  This process
 3:    manages the files located on its node. }
 4:
 5:  var
 6:     msg: message_pointer;
 7:     file_directory: file_location_information;
 8:
 9:  begin
10:     loop
11:        <wait for the arrival of a message (let msg point
12:           to the message)>;
13:        case msg^.message_type of
14:           M8: { file availability request }
15:              begin
16:                 for <each file named in msg^> do
17:                    <search for the file>;
18:                    if <the file was found> then
19:                       if <the file is free> then
20:                          <reserve the file>;
21:                          <record the desired access to the file>;
22:                          <note that the file is available>;
23:                       else
24:                          if <the desired access to the file
25:                             is READ> and <the access already
26:                             granted to the file is READ> then
27:                             <note that the file is available>;
28:                          else
29:                             <note that the file is not available>;
30:                          endif;
31:                       endif;
32:                    else
33:                       <note that the file is not available>;
34:                    endif;
35:                 endfor;
36:                 <send a message of type M12 (file availability
37:                    information) to the file_system_manager
38:                    on node msg^.node>;
39:              end;
40:           M9: { file lock and release request }
41:              begin
42:                 for <each file in msg^> do
43:                    <search for the file>;
44:                    if <the file was found> then
45:                       <lock or release the file as requested>;
46:                    else
47:                       <note that the request could not be satisfied>;
48:                    endif;
49:                 endfor;
```

```
50:                  <send a message of type M13 (results of file lock
51:                     and release request) to the file_system_manager
52:                     on node msg^.node>;
53:               end;
54:            M10: { file release request }
55:               begin
56:                  for <each file in msg^> do
57:                     <search for the file and release the lock on it>;
58:                  endfor;
59:               end;
60:            M11: { file creation request }
61:               begin
62:                  <create an entry for a new file in file_directory>;
63:                  <activate a file process for the file>;
64:                  <send a message of type M14 (results of file
65:                     creation) to the file_system_manager on
66:                     node msg^.node>;
67:               end;
68:         endcase;
69:      endloop;
70:   end file_set_manager;
```

1.2 PSEUDO CODE FOR THE XFDPS.2 CONTROL MODEL

1.2.1 System Initiator

Same as XFDPS.1.

1.2.2 Task Set Manager

XFDPS.1 with the following changes:

```
25:   <send a message of type M2 (file availability request) to
26:      the file_system_manager on node 1 that contains the
27:      names of files needed for this work request>;

44:   <send a message of type M3 (file lock and release request)
45:      to the file_system_manager on node 1>;

76:   <send a message of type M4 (file release request)
77:      to the file_system_manager on node 1>;
```

1.2.3 File System Manager

```
process file_system_manager;
{ This process resides on node 1 and satisfies various requests
  concerning the file system.  This process maintains the
  centralized file system directory. }

var
   msg: message_pointer;
```

```
begin
   loop
      <wait for a message of any type (let msg point to
         the message)>;
      case msg^.message_type of
         M1: { file availability information request }
            begin
               for <each file named in msg^> do
                  <search for the file>;
                  if <the file was found> then
                     for <each node i> do
                        if <the file is free on node i> then
                           <reserve the file>;
                           <record the desired access to the file>;
                           <note that the file is available on
                              node i>;
                        else
                           if <the desired access to the file
                              is READ> and <the access already
                              granted to the file is READ> then
                              <note that the file is available on
                                 node i>;
                           else
                              <note that the file is not available
                                 on node i>;
                           endif;
                        endif;
                     endfor;
                  else
                     <note that the file is not available on
                        any node>;
                  endif;
               endfor;
               <send a message of type M12 (file availability
                  information) to the task_set_manager requesting
                  the information>;
            end;
         M3: { file lock and release request }
            begin
               for <each file in msg^> do
                  <search for the file>;
                  if <the file was found and is present
                     on the node specified> then
                     <lock or release the file as requested>;
                  else
                     <note that the request could not be satisfied>;
                  endif;
               endfor;
               <send a message of type M13 (results of file lock
                  and release request) to the task_set_manager
                  that made the request>;
            end;
```

```
         M4: { file release request }
            begin
               for <each file in msg^> do
                  <search for the file and release the lock on it>;
               endfor;
            end;
      endcase;
   endloop;
end file_system_manager;
```

1.2.4 Process Utilization Manager

Same as XFDPS.1.

1.2.5 Processor Utilization Monitor

Same as XFDPS.1.

1.2.6 Process Manager

Same as XFDPS.1.

1.3 PSEUDO CODE FOR THE XFDPS.3 CONTROL MODEL

1.3.1 System Initiator

Same as XFDPS.1.

1.3.2 Task Set Manager

Same as XFDPS.1.

1.3.3 File System Manager

XFDPS.1 with the following changes:

```
23:   <send a message of type M8 (file availability
24:      request) to the file_set_manager on the same node
25:      as this file_system_manager>;
26:
27:

69:   if <this response is from this node> and
70:      <all files have not been found available> then
71:      for <every other node i> do
72:         <send a message of type M8 (file availability
73:            request) to the file_set_manager on node i>;
74:      endfor;
74a:  else
74b:     if <responses from all file_set_managers have been
74c:        received or all files have been found locally> then
```

```
74d:        <send a message of type M16 (file availability
74e:           information) to the task_set_manager identified
74f:           by a field of favptr^>;
74g:     endif;
74h:  endif;
```

1.3.4 Process Utilization Manager

Same as XFDPS.1.

1.3.5 Processor Utilization Monitor

Same as XFDPS.1.

1.3.6 Process Manager

Same as XFDPS.1.

1.6.7 File Set Manager

Same as XFDPS.1.

1.4 PSEUDO CODE FOR THE XFDPS.4 CONTROL MODEL

1.4.1 System Initiator

Same as XFDPS.1.

1.4.2 Task Set Manager

Same as XFDPS.1.

1.4.3 File System Manager

```
process file_system_manager;
{ Every node possesses one of these processes.  This process
  satisfies various requests concerning the file system and
  helps maintain the redundant copies of the file system
  directory. }

var
   msg: message_pointer;

begin
   loop
      <wait for a message of any type (let msg point to
         the message)>;
      case msg^.message_type of
         M1, M3, M4: { availability, lock, and release requests }
            begin
               <place the message on the queue of file system
                  requests arriving at this node>;
            end;
```

```
CV: { control vector }
   begin
      while <the file system request queue is
         not empty> do
         <remove a message from the queue (let msg point
            to the message)>;
         case msg^.message_type of
            M1: { file availability information request }
               begin
                  for <each file named in msg^> do
                     <search for the file>;
                     if <the file was found> then
                        for <each node i> do
                           if <the file is free on node i> then
                              <reserve the file>;
                              <record the desired access to the file>;
                              <note that the file is available on
                                 node i>;
                           else
                              if <the desired access to the file
                                 is READ> and <the access already
                                 granted to the file is READ> then
                                 <note that the file is available on
                                    node i>;
                              else
                                 <note that the file is not available
                                    on node i>;
                              endif;
                           endif;
                        endfor;
                     else
                        <note that the file is not available on
                           any node>;
                     endif;
                  endfor;
                  <send a message of type M12 (file availability
                     information) to the task_set_manager requesting
                     the information>;
               end;
            M3: { file lock and release request }
               begin
                  for <each file in msg^> do
                     <search for the file>;
                     if <the file was found and is present
                        on the node specified> then
                        <lock or release the file as requested>;
                     else
                        <note that the request could not be satisfied>;
                     endif;
                  endfor;
```

```
                        <send a message of type M13 (results of file lock
                           and release request) to the task_set_manager
                           that made the request>;
                     end;
                  M4: { file release request }
                     begin
                        for <each file in msg^> do
                           <search for the file and release the lock on it>;
                        endfor;
                     end;
               endcase;
            endwhile;
            <send a message of type UPV (update vector) to the
               next node (according to the predetermined
               ordering of nodes) containing the changes just
               made to the file system directory>;
         end;
      UPV: { update vector }
         begin
            if <this UPV was originated by this node> then
               <send a message of type CV (control vector) to
                  the next node (according to the predetermined
                  ordering of nodes)>;
            else
               <update the file system directory>;
               <send the message of type UPV (update vector)
                  to the next node (according to the predetermined
                  ordering of nodes)>;
            endif;
         end;
   endcase;
endloop;
end file_system_manager;
```

1.4.4 Process Utilization Manager

Same as XFDPS.1.

1.4.5 Processor Utilization Monitor

Same as XFDPS.1.

1.4.6 Process Manager

Same as XFDPS.1.

1.5 PSEUDO CODE FOR THE XFDPS.5 CONTROL MODEL

1.5.1 System Initiator

Same as XFDPS.1.

1.5.2 Task Set Manager

Same as XFDPS.1.

1.5.3 File System Manager

Same as XFDPS.1.

1.5.4 Process Utilization Manager

Same as XFDPS.1.

1.5.5 Processor Utilization Monitor

Same as XFDPS.1.

1.5.6 Process Manager

Same as XFDPS.1.

1.5.7 File Set Manager

XFDPS.1 with the following changes:

```
20:   <note that the file is available>;
21:
22:
```

1.6 PSEUDO CODE FOR THE XFDPS.6 CONTROL MODEL

1.6.1 System Initiator

Same as XFDPS.1.

1.6.2 Task Set Manager

XFDPS.1 with the following changes:

```
75:   for <each task in the message> do
76:      <mark the task as completed in tg^>;
77:   endfor;

87:   for <every node i still executing parts of the work
88:      request> do
89:      <send a message of type M7 (process kill request)
90:         to the process_manager on node i>;
91:   endfor;
92:
93:
```

1.6.3 File System Manager

Same as XFDPS.1.

1.6.4 Process Utilization Manager

Same as XFDPS.1.

1.6.5 Processor Utilization Monitor

Same as XFDPS.1.

1.6.6 Process Manager

```
process process_manager;
{ Every node possesses one of these processes.  This process
  manages the processes that are executing on its node. }

var
   pcbptr: process_control_block_pointer;
   process_name_table: process_name_to_pcbptr_map;
   subtg: task_graph_pointer;
   msg: message_pointer;

begin
   loop
      <wait for the arrival of a message (let msg point
         to the message)>;
      case msg^.message_type of
         M6: { process activation request }
            begin
               new (subtg);
               for <each task i im msg^> do
                  <record task i in subtg^>;
                  if <task i names an object file> then
                     new (pcbptr);
                     <record process identifying information
                        and pcbptr in process_name_table>;
                     <fill in the necessary information in pcbptr^>;
                     <initiate the loading of the process>;
                  else
                     task_set_manager (CMNDFILE, msg^.file_descriptor);
                     <record process identifying information
                        and task_set_manager identification in
                        process_name_table>;
                  endif;
               endfor;
               <link subtg^ onto the list of subtaskgraphs executing
                  on this node>;
            end;
```

```
        M7: { process kill request }
            begin
               <find the subtaskgraph in the list of
                  subtaskgraphs executing on this node (let
                  subtg point to the subtaskgraph)>;
               for <each task i in subtg^> do
                  if <task i has not completed> then
                     if <task i names an object file> then
                        <terminate the process>;
                        <unload the process>;
                        <dispose of the process control block>;
                        <mark task i as terminated>;
                     else { the process is a command file }
                        <send a message of type M25 (request to terminate
                           the execution of a command file) to the
                           task_set_manager executing this command file>;
                     endif;
                  endif;
               endfor;
               if <all the tasks in subtg^ have completed> then
                  <send a message of type M24 (subtaskgraph
                     termination message) to the task_set_manager
                     that activated the subtaskgraph>;
                  <remove subtg^ from the list of subgraphs
                     executing on this node>;
                  dispose (subtg);
               endif;
            end;
      endcase;
   endloop;
end process_manager;
```

APPENDIX 2

SIMULATION RESULTS

2.1 RESULTS OF GROUP 1 EXPERIMENTS

2.1.1 XFDPS.1

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside				
	1	2	3	4	5
1200	6.634	7.123	7.123	7.123	7.123
50,000	6.306	6.320	6.320	6.320	6.320
100,000	6.302	6.310	6.310	6.310	6.310
500,000	6.299	6.303	6.303	6.303	6.303

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside				
	1	2	3	4	5
1200	6.506	6.743	6.911	6.911	6.743
50,000	6.302	6.308	6.313	6.313	6.308
100,000	6.299	6.303	6.306	6.306	6.303
500,000	6.298	6.299	6.301	6.301	6.299

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside				
	1	2	3	4	5
1200	6.380	6.617	6.617	6.617	6.617
50,000	6.298	6.304	6.304	6.304	6.304
100,000	6.297	6.301	6.301	6.301	6.301
500,000	6.297	6.299	6.299	6.299	6.299

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside				
	1	2	3	4	5
1200	6.380	6.617	6.617	6.617	6.617
50,000	6.298	6.304	6.304	6.304	6.304
100,000	6.297	6.301	6.301	6.301	6.301
500,000	6.297	6.299	6.299	6.299	6.299

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.549	6.786	6.786	6.954	6.954
50,000	6.303	6.309	6.309	6.314	6.314
100,000	6.300	6.303	6.303	6.307	6.307
500,000	6.298	6.300	6.300	6.301	6.301

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 2

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.123	6.634	7.123	7.123	7.123
50,000	6.320	6.306	6.320	6.320	6.320
100,000	6.310	6.302	6.310	6.310	6.310
500,000	6.303	6.299	6.303	6.303	6.303

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.743	6.506	6.743	6.911	6.911
50,000	6.308	6.302	6.308	6.313	6.313
100,000	6.303	6.299	6.303	6.306	6.306
500,000	6.299	6.298	6.299	6.301	6.301

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.828	6.591	6.997	6.997	6.997
50,000	6.310	6.304	6.315	6.315	6.315
100,000	6.304	6.301	6.307	6.307	6.307
500,000	6.300	6.299	6.302	6.302	6.302

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.617	6.380	6.617	6.617	6.617
50,000	6.304	6.298	6.304	6.304	6.304
100,000	6.301	6.297	6.301	6.301	6.301
500,000	6.299	6.297	6.299	6.299	6.299

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.743	6.506	6.911	6.743	6.743
50,000	6.307	6.302	6.313	6.307	6.307
100,000	6.303	6.299	6.306	6.303	6.303
500,000	6.299	6.298	6.301	6.299	6.299

2.1.2 XFDPS.2

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.784	6.784	6.784	6.784
50,000	6.295	6.308	6.308	6.309	6.309
100,000	6.295	6.303	6.303	6.303	6.303
500,000	6.295	6.299	6.299	6.299	6.299

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.532	6.700	6.700	6.532
50,000	6.295	6.301	6.306	6.306	6.301
100,000	6.295	6.298	6.301	6.301	6.298
500,000	6.295	6.296	6.298	6.298	6.296

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.532	6.532	6.532	6.532
50,000	6.295	6.301	6.301	6.301	6.301
100,000	6.295	6.298	6.298	6.298	6.298
500,000	6.295	6.296	6.296	6.296	6.296

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.532	6.532	6.532	6.532
50,000	6.295	6.301	6.301	6.301	6.301
100,000	6.295	6.298	6.298	6.298	6.298
500,000	6.295	6.296	6.296	6.296	6.296

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.532	6.532	6.700	6.700
50,000	6.295	6.301	6.301	6.306	6.306
100,000	6.295	6.298	6.298	6.301	6.301
500,000	6.295	6.296	6.296	6.298	6.298

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 2

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.951	6.966	7.203	7.203	7.203
50,000	6.313	6.314	6.321	6.321	6.321
100,000	6.306	6.306	6.310	6.310	6.310
500,000	6.300	6.300	6.302	6.302	6.302

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.573	6.588	6.783	6.867	6.783
50,000	6.302	6.303	6.308	6.310	6.308
100,000	6.299	6.299	6.302	6.304	6.302
500,000	6.296	6.296	6.298	6.299	6.298

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.573	6.588	6.783	6.783	6.783
50,000	6.302	6.303	6.308	6.308	6.308
100,000	6.299	6.299	6.302	6.302	6.302
500,000	6.296	6.296	6.298	6.298	6.298

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.573	6.588	6.699	6.699	6.699
50,000	6.302	6.303	6.305	6.305	6.305
100,000	6.299	6.299	6.301	6.301	6.301
500,000	6.296	6.296	6.297	6.297	6.297

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.573	6.588	6.783	6.783	6.783
50,000	6.302	6.303	6.308	6.308	6.308
100,000	6.299	6.299	6.302	6.302	6.302
500,000	6.296	6.296	6.298	6.298	6.298

2.1.3 XFDPS.3

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	7.123	7.123	7.123	7.123
50,000	6.295	6.320	6.320	6.320	6.320
100,000	6.295	6.311	6.311	6.311	6.311
500,000	6.295	6.303	6.303	6.303	6.303

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.743	6.912	6.912	6.743
50,000	6.295	6.308	6.313	6.313	6.308
100,000	6.295	6.303	6.306	6.306	6.303
500,000	6.295	6.299	6.301	6.301	6.299

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.617	6.617	6.617	6.617
50,000	6.295	6.305	6.305	6.305	6.305
100,000	6.295	6.301	6.301	6.301	6.301
500,000	6.295	6.299	6.299	6.299	6.299

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.617	6.617	6.617	6.617
50,000	6.295	6.305	6.305	6.305	6.305
100,000	6.295	6.301	6.301	6.301	6.301
500,000	6.295	6.299	6.299	6.299	6.299

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.295	6.786	6.786	6.954	6.954
50,000	6.295	6.309	6.309	6.315	6.315
100,000	6.295	6.304	6.304	6.307	6.307
500,000	6.295	6.300	6.300	6.301	6.301

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 2

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.123	6.295	7.123	7.123	7.123
50,000	6.320	6.295	6.320	6.320	6.320
100,000	6.311	6.295	6.311	6.311	6.311
500,000	6.303	6.295	6.303	6.303	6.303

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.744	6.295	6.744	6.912	6.912
50,000	6.308	6.295	6.308	6.314	6.314
100,000	6.303	6.295	6.303	6.306	6.306
500,000	6.299	6.295	6.299	6.301	6.301

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.829	6.295	6.997	6.997	6.997
50,000	6.311	6.295	6.316	6.316	6.316
100,000	6.304	6.295	6.308	6.308	6.308
500,000	6.300	6.295	6.302	6.302	6.302

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.617	6.295	6.617	6.617	6.617
50,000	6.305	6.295	6.305	6.305	6.305
100,000	6.301	6.295	6.301	6.301	6.301
500,000	6.299	6.295	6.299	6.299	6.299

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.743	6.295	6.912	6.743	6.743
50,000	6.308	6.295	6.313	6.308	6.308
100,000	6.303	6.295	6.306	6.303	6.303
500,000	6.299	6.295	6.301	6.299	6.299

2.1.4 XFDPS.A

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.925	7.219	7.219	7.219	7.220
50,000	6.360	6.367	6.368	6.368	6.368
100,000	6.353	6.358	6.358	6.358	6.358
500,000	6.316	6.318	6.318	6.318	6.318

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.925	7.093	7.177	7.135	7.051
50,000	6.360	6.364	6.366	6.366	6.364
100,000	6.353	6.355	6.357	6.357	6.354
500,000	6.316	6.317	6.318	6.317	6.316

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.302	7.471	7.428	7.428	7.441
50,000	6.376	6.355	6.354	6.354	6.354
100,000	6.376	6.339	6.339	6.339	6.338
500,000	6.581	6.357	6.323	6.322	6.356

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.925	7.093	7.051	7.051	7.051
50,000	6.360	6.364	6.363	6.363	6.364
100,000	6.353	6.355	6.355	6.354	6.354
500,000	6.316	6.317	6.316	6.316	6.316

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.555	7.723	7.681	7.807	7.806
50,000	6.373	6.367	6.351	6.354	6.354
100,000	6.482	6.470	6.338	6.340	6.340
500,000	6.385	6.448	6.354	6.335	6.335

RESPONSE TIME (sec) FOR A SINGLE WORK REQUEST ARRIVING AT NODE 2

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.051	6.757	7.051	7.051	7.051
50,000	6.362	6.354	6.362	6.362	6.362
100,000	6.354	6.350	6.354	6.354	6.354
500,000	6.317	6.314	6.316	6.316	6.316

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.883	6.757	6.925	7.009	6.967
50,000	6.358	6.354	6.358	6.361	6.360
100,000	6.350	6.350	6.352	6.354	6.353
500,000	6.315	6.314	6.315	6.316	6.315

STAR

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.177	7.008	7.260	7.303	7.260
50,000	6.370	6.342	6.348	6.348	6.348
100,000	6.371	6.330	6.335	6.335	6.336
500,000	6.580	6.319	6.355	6.321	6.320

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	6.883	6.757	6.925	6.883	6.882
50,000	6.358	6.354	6.358	6.358	6.357
100,000	6.350	6.350	6.352	6.351	6.351
500,000	6.315	6.314	6.315	6.315	6.315

TREE

Bandwidth (bytes/sec)	Node Where Object and Data Files Reside 1	2	3	4	5
1200	7.345	7.177	7.428	7.302	7.302
50,000	6.355	6.351	6.342	6.339	6.338
100,000	6.474	6.461	6.332	6.330	6.330
500,000	6.381	6.444	6.331	6.330	6.330

2.2 RESULTS OF GROUP 2 EXPERIMENTS

2.2.1 XFDPS.1

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node 1	2	3	4	5
50	182.3	1.0	1.0	1.0	1.0	1.2
100	169.7	1.9	1.1	1.7	2.8	1.3
200	92.8	2.8	1.9	2.7	4.4	5.5
600	47.9	6.0	7.9	5.0	14.6	5.4
1200	45.0	13.1	12.5	5.5	2.1	5.2
50,000	48.2	18.4	2.4	7.1	10.0	2.8
100,000	41.6	7.1	6.0	10.4	6.6	7.0
500,000	35.7	5.4	15.4	4.6	8.5	2.3
2,500,000	42.2	8.0	10.1	12.0	7.2	2.3

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node 1	2	3	4	5
50	109.4	1.3	1.5	1.2	1.8	1.1
100	57.6	6.2	4.7	3.4	3.5	4.1
200	48.8	4.1	15.3	3.7	9.4	3.9
600	44.2	8.7	3.7	8.4	1.5	15.7
1200	40.5	7.3	6.6	6.4	5.0	11.4
50,000	43.3	10.0	15.0	4.0	4.5	4.0
100,000	47.5	10.5	6.2	5.6	11.3	9.6
500,000	42.5	7.3	12.4	4.8	10.8	5.9
2,500,000	47.7	5.6	7.3	8.3	17.8	3.6

STAR

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	133.2	2.5	1.1	1.0	1.1	2.3
100	66.4	4.0	4.9	2.4	1.3	3.9
200	44.3	4.7	13.9	4.4	4.1	2.5
600	46.8	3.6	18.3	11.8	3.1	1.9
1200	46.5	5.2	8.7	5.7	8.5	12.0
50,000	41.4	5.3	7.2	11.2	7.5	7.5
100,000	45.0	3.5	19.1	6.4	6.0	4.1
500,000	39.9	5.2	11.8	4.9	11.5	3.5
2,500,000	43.0	9.2	11.9	6.7	4.7	7.2

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	47.7	10.3	2.2	4.5	2.6	3.8
100	43.8	11.4	9.9	3.2	5.3	2.9
200	46.7	1.9	5.6	13.5	15.0	2.8
600	42.6	4.0	4.7	7.1	11.1	11.5
1200	43.2	8.2	11.7	6.7	9.5	3.3
50,000	44.0	12.1	14.9	5.3	3.1	5.2
100,000	44.4	3.2	17.1	4.3	8.3	7.5
500,000	42.8	6.5	4.2	4.2	11.0	12.5
2,500,000	41.3	11.9	3.9	7.5	5.3	8.9

TREE

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	190.4	1.0	1.2	1.2	1.0	1.0
100	93.4	2.0	1.6	1.2	2.6	2.0
200	51.0	6.9	3.8	8.8	3.0	1.5
600	47.9	7.9	9.0	13.9	7.3	3.8
1200	44.4	10.5	9.9	4.2	6.1	7.8
50,000	44.5	8.0	4.5	10.1	10.5	5.3
100,000	46.4	9.6	1.8	16.4	10.0	5.0
500,000	43.3	9.8	7.2	13.2	5.1	3.6
2,500,000	45.4	12.8	6.1	4.8	7.4	10.3

2.2.2 XFDPS.2

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	210.3	1.0	2.8	1.9	1.0	1.0
100	103.1	1.7	4.1	2.0	1.6	1.5
200	53.7	1.9	3.4	2.6	3.4	3.9
600	41.3	9.9	7.7	3.4	10.7	4.2
1200	47.1	1.8	12.3	20.4	3.1	2.7
50,000	44.9	4.5	4.5	15.9	11.7	4.5
100,000	47.4	2.5	2.9	19.3	12.2	3.7
500,000	49.4	3.2	7.0	9.2	19.6	2.1
2,500,000	45.4	8.0	5.9	4.2	7.0	16.4

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	93.3	1.1	1.2	2.5	1.6	1.3
100	63.1	1.7	6.3	16.2	4.9	1.1
200	48.1	6.6	9.7	3.4	4.7	14.4
600	41.5	7.3	7.9	7.0	10.2	3.9
1200	43.1	3.2	5.0	5.7	5.1	18.7
50,000	41.7	4.6	16.9	4.2	5.2	9.8
100,000	43.1	2.8	11.9	8.3	5.1	11.5
500,000	44.0	12.7	8.8	5.7	4.5	7.0
2,500,000	51.3	4.4	3.7	13.3	19.2	4.3

STAR

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	58.7	1.8	3.4	1.7	2.2	1.9
100	43.0	3.7	7.3	9.5	6.6	2.9
200	45.0	3.2	5.6	5.4	17.9	4.6
600	45.4	5.2	18.8	5.9	4.6	5.8
1200	41.9	9.7	13.1	3.8	5.5	5.0
50,000	43.5	4.3	20.1	3.3	4.3	4.9
100,000	45.9	3.5	10.7	17.1	5.6	4.4
500,000	46.2	5.3	6.2	4.8	20.7	4.9
2,500,000	40.9	1.9	8.4	3.0	20.9	4.6

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	47.6	2.3	5.8	2.4	2.8	2.2
100	51.4	3.3	1.7	7.4	14.4	12.7
200	47.0	3.4	19.3	10.3	4.1	2.5
600	42.9	2.2	8.8	6.0	10.6	10.6
1200	46.3	9.2	7.4	3.7	4.3	17.3
50,000	39.7	7.6	7.3	4.2	4.4	12.6
100,000	38.2	5.9	18.3	3.8	4.1	6.6
500,000	46.1	9.2	13.8	4.6	9.2	3.2
2,500,000	49.2	20.3	8.0	3.8	6.7	2.8

TREE

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	132.7	1.0	1.0	1.2	2.2	1.0
100	66.0	1.5	5.5	2.1	1.5	1.1
200	45.4	7.0	4.0	6.7	12.5	2.8
600	43.9	6.0	10.1	5.8	2.5	14.2
1200	45.5	11.1	5.2	10.6	8.8	4.6
50,000	42.0	3.5	10.9	12.5	6.8	4.8
100,000	42.1	6.8	7.2	17.3	7.1	3.0
500,000	45.6	18.3	6.0	2.9	9.1	3.4
2,500,000	48.2	5.1	7.1	1.9	24.2	2.9

2.2.3 XFDPS.3

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
100	141.3	1.6	1.9	1.3	3.7	5.0
200	82.5	4.2	2.0	1.2	2.8	3.1
600	45.2	11.5	13.5	2.2	6.1	3.7
1200	43.6	10.4	18.8	3.4	2.0	3.6
50,000	39.1	7.6	4.2	7.3	9.0	9.9
100,000	43.5	13.0	5.4	12.2	4.6	3.4
500,000	45.6	14.5	8.1	2.4	2.7	11.6
2,500,000	45.2	2.8	7.2	5.6	8.2	16.2

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	99.9	1.9	1.1	1.4	1.1	2.2
100	54.4	5.0	2.5	2.9	5.1	11.0
200	45.9	19.1	2.6	6.4	3.8	4.9
600	40.4	7.2	4.6	4.4	5.9	14.8
1200	49.2	11.7	12.3	6.1	7.2	4.4
50,000	39.3	4.7	7.3	12.4	9.7	4.2
100,000	40.4	1.7	4.8	5.4	19.1	8.2
500,000	47.9	4.9	15.6	13.8	3.4	3.6
2,500,000	42.8	15.8	3.1	4.6	8.0	8.9

STAR

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	114.5	1.6	1.9	1.0	1.5	1.6
100	59.4	3.0	2.7	2.2	5.1	1.9
200	45.9	5.8	9.7	11.5	3.7	4.9
600	39.9	10.6	13.6	5.2	5.0	2.8
1200	39.5	2.7	12.8	13.6	4.3	3.2
50,000	45.9	5.0	24.7	4.4	4.2	2.9
100,000	44.7	3.3	9.6	12.1	7.4	8.5
500,000	44.9	9.8	8.4	4.4	4.7	15.0
2,500,000	36.2	7.3	4.7	15.5	7.4	3.0

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	47.2	3.0	6.7	2.2	9.2	4.6
100	42.8	3.5	5.5	6.5	9.6	8.1
200	44.5	3.8	7.8	8.5	12.2	6.4
600	47.2	14.2	7.3	11.2	3.4	4.9
1200	45.1	13.4	16.7	3.6	2.4	4.1
50,000	39.9	4.9	22.4	3.9	3.5	4.0
100,000	36.3	9.0	6.5	6.7	7.6	5.9
500,000	43.1	4.1	11.1	13.0	7.4	3.5
2,500,000	43.6	4.4	7.4	17.9	4.1	5.2

TREE

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	154.6	1.0	1.1	1.2	1.0	1.0
100	95.1	1.0	1.4	4.5	1.2	1.0
200	47.7	2.2	3.9	4.8	2.7	19.6
600	47.0	14.7	2.7	2.5	8.9	10.4
1200	45.7	10.1	9.2	5.5	9.7	8.0
50,000	43.9	4.6	9.2	4.6	16.2	5.1
100,000	43.3	9.4	8.0	4.4	6.7	10.1
500,000	45.0	10.1	6.0	6.8	4.3	11.4
2,500,000	43.8	3.4	10.7	12.3	7.0	5.9

2.2.4 XFDPS.4

UNIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	111.4	2.5	1.0	1.2	1.5	1.4
100	63.0	7.8	2.2	1.3	2.4	2.9
200	48.5	3.9	2.7	3.3	9.2	9.9
600	45.6	21.1	2.3	4.3	3.0	7.1
1200	48.3	4.2	3.1	15.2	14.2	3.2
50,000	45.2	10.5	5.0	14.8	4.0	5.7
100,000	48.1	6.2	9.0	8.1	5.2	14.3
500,000	46.3	5.3	8.2	6.6	16.3	4.9
2,500,000	44.4	6.9	5.1	11.2	9.4	6.5

BIDIRECTIONAL RING

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	80.4	4.4	1.7	1.9	2.8	1.5
100	56.2	4.4	4.2	6.5	2.8	6.8
200	49.1	8.5	14.3	2.6	3.9	2.3
600	44.9	9.5	6.0	4.9	11.1	4.9
1200	45.5	3.0	11.3	11.5	4.3	10.3
50,000	38.6	6.2	6.1	8.0	9.2	6.1
100,000	38.8	6.3	8.5	5.9	9.1	6.4
500,000	44.9	6.9	9.6	7.0	9.8	8.6
2,500,000	43.0	1.8	4.0	18.7	7.0	7.9

STAR

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	125.0	1.1	1.6	1.3	1.3	1.3
100	64.7	4.8	4.0	4.2	1.7	1.9
200	53.6	5.4	6.9	2.9	9.8	3.9
600	46.2	6.1	7.0	5.5	8.1	8.3
1200	44.4	9.8	6.8	3.7	4.4	14.1
50,000	40.7	5.3	6.7	6.7	6.7	11.1
100,000	44.9	4.4	5.7	11.6	3.7	14.1
500,000	48.1	21.0	8.4	4.0	5.5	1.9
2,500,000	43.8	3.1	12.7	8.8	2.3	12.2

FULLY CONNECTED NETWORK

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	68.3	3.9	1.2	1.7	5.4	3.5
100	51.3	2.5	2.4	7.6	6.0	2.8
200	47.3	6.7	5.0	2.7	13.2	3.1
600	47.4	21.7	7.5	3.6	2.5	3.6
1200	43.3	3.5	14.5	8.4	7.6	3.7
50,000	44.2	6.5	16.0	11.6	3.2	1.4
100,000	45.4	3.7	8.0	17.2	6.4	5.2
500,000	43.5	6.3	7.6	8.7	6.0	11.2
2,500,000	41.2	4.4	4.3	4.3	10.5	14.2

TREE

Bandwidth (bytes/sec)	Ave Response Time (sec)	Average Length of the READY QUEUE on Each Node				
		1	2	3	4	5
50	134.8	2.1	1.0	1.3	1.7	1.2
100	72.8	1.4	3.5	3.4	3.3	1.9
200	52.2	7.3	6.0	3.9	3.5	3.1
600	45.8	5.4	8.4	9.9	5.2	3.7
1200	46.3	4.4	8.0	3.6	15.9	4.2
50,000	43.5	3.9	7.5	12.3	6.5	9.6
100,000	36.3	3.2	8.7	5.3	7.3	9.8
500,000	42.2	3.0	21.1	4.4	4.3	7.4
2,500,000	45.0	7.1	10.7	11.6	6.5	3.8

2.3 RESULTS OF A SINGLE NODE SIMULATION

Average Work Request Response Time for
a Single Node Network

Run	Average Response Time (sec)
1	44.6
2	44.1
3	43.7
4	43.7
5	44.2

Mean: 44.1 seconds

Standard Deviation: 0.38